Deep Clustering for Metagenomic Binning

Jan Niklas Fichte, Trong Dai Ha, Jitka Polaskova

Computer Science, cs-22-mi-10-02, 2022-06

Master Thesis

Computer Science

Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Deep Clustering for Metagenomic Binning

Theme:
Master Thesis

Project Period:
Spring Semester 2022
01. February 2022 - 17. June 2022

Project Group:
cs-22-mi-10-02

Participants:
- Jan Niklas Fichte
jficht20@student.aau.dk
- Trong Dai Ha
dha19@student.aau.dk
- Jitka Polaskova
jpolas20@student.aau.dk

Supervisor:
- Katja Hose
khose@cs.aau.dk
- Thomas Dyhre Nielsen
tdn@cs.aau.dk

Page Numbers:
117

Date of Completion:
June 17, 2022

Abstract:

Deep learning is an area that is only sparsely explored for metagenomic binning. The existing deep learning-based approaches usually preprocess raw DNA sequences into input features such as composition and abundance and perform representation learning and clustering in two steps. The utilization of unprocessed DNA sequences as input shows promising results for gene prediction. Joint deep clustering leads to better results for image clustering than basic approaches like k-means clustering. In this report, we investigate the potential of joint end-to-end unsupervised learning and the utilization of unprocessed contigs as inputs for the task of metagenomic binning.
We propose two new binners: ***Deep Convolutional Metagenomic Binner*** (DCMB) and ***Deep Stacked Metagenomic Binner*** (DSMB). Both binners utilize KL divergence-based joint deep clustering. The DCMB takes unprocessed contigs and the DSMB uses abundance and composition as inputs.
The performance of both binners is benchmarked on the CAMI Low dataset and compared to metagenomic binners VAMB, MetaBat2, and SolidBin. The results show that metagenomic information requires preprocessing to obtain meaningful representations and that joint end-to-end learning slightly improves the number of recovered bins.

Acknowledgements

We want to thank our supervisors, Katja Hose and Thomas Dyhre Nielsen, for their great input and feedback throughout the previous project and this project. Further, we want to thank André Lamúrias for guiding us through our questions.

Summary

State-of-the-art metagenomic binners usually preprocess raw DNA sequences into input features like composition and abundance [1][2]. Moreover, existing binners based on deep learning learn a lower-dimensional representation of the data in one step and cluster on it in a second step.
We aim to explore the impact of joint deep clustering in metagenomic binning and to investigate if it is possible to get meaningful results by utilizing unprocessed contigs instead of abundance and composition features. Joint deep clustering means that representation learning and clustering are optimized simultaneously.
We started with an investigation of methods and techniques that are useful to achieve our objectives. First, we describe the bioinformatical preliminaries, including what metagenomics and sequencing are. To not require to process the input, we want to take unprocessed contigs as input instead of composition and abundance. We outline which information composition and abundance abstracts and how both features are obtained to better understand the usage of these features. To understand the possibilities that exist to process contigs, we explore the basics of deep neural networks and different deep neural network architectures. The investigated methods are based on convolutional neural networks and different variants of autoencoders.
Further, we explore different deep clustering categories that exist and investigate deeper in the area of joint deep clustering. Then, we explore and compare the different state-of-the-art joint deep clustering frameworks to find a method that is suitable in terms of architecture and resources for metagenomic binning. The frameworks that we explore are Joint Unsupervised Learning (JULE), Deep Embedded Regularized Clustering (DEPICT), Deep Convolutional Embedded Clustering (DCEC) and Deep Embedding Clustering (DEC).
Based on the knowledge obtained through the background, we propose two metagenomic binners: Deep Convolutional Metagenomic Binner (DCMB) and Deep Stacked Metagenomic Binner (DSMB). Both binners utilize joint deep clustering and use KL divergence clustering as a clustering technique. DCMB is based on convolutional autoencoders and takes unprocessed contigs as input. The raw DNA sequences

are one-hot encoded, padded, and trimmed to a common length. DSMB is based on stacked autoencoders and takes composition and abundance as input.
To validate the performance of the two binners, we benchmark them against the state-of-the-art binners VAMB [1], MetaBat2 [2], and SolidBin [3] on the CAMI Low dataset [4]. DSMB is additionally benchmarked against the mentioned binners on the Azolla dataset [5]. With further experiments, we explore the impact of utilizing unprocessed contigs and joint deep clustering. The benchmark showed that DCMB does not recover any genome as assessed with AMBER. The number of genomes recovered by DSMB is close to the number of VAMB and SolidBin. By comparing the DSMB with a variant that does representation learning and clustering in two steps instead of one joint step, we found out that the joint deep clustering has a slight impact. One to two bins more are recovered if using joint learning.
In general, we observed on DCMB that unprocessed contigs alone may not be sufficient to obtain meaningful bins. One of our future works includes combining feature representation learned from raw contigs together with abundance feature as an input of a metagenomic binner. DSMB showed that joint end-to-end learning bears potential, and with another joint end-to-end clustering architecture it might be possible to recover even more bins.

Contents

Chapter 1

Introduction

Metagenomic binning is a process of grouping metagenomic sequences by their organism of origin [1]. The aim of it is to identify which known and unknown organisms are present in an environmental sample. Environmental samples can be a small samples of tissue, water, soil or surface that were collected for the purpose of analysis [6]. Among other things, this may provide an insight into which organisms share a habitat [7] and give an insight into biological processes. These insights are important as they help to better understand the environment and to make use of these processes. An example is the rice production the free-floating water fern Azolla is used as fertilizer. Azolla lives in symbiosis with a Nitrogen fixing cyanobacterium species [8].

The first step in metagenomic binning is to identify DNA (deoxyribonucleic acid) sequences of the organisms in the environmental sample. The DNA contains the genetic information of an organism. DNA is composed of nucleotides, and the four nucleobases are adenine (A), guanine (G), thymine (T) and cytosine (C). The DNA molecule is structured as a double helix. Connected are the two strands through base pairs. Adenine and thymine form one pair, and the other one is guanine and cytosine [9]. The DNA-sequences are identified by devices that are called sequencers. A sequencer determines the sequences of nucleotides and groups them into so-called reads. Reads are smaller DNA fragments which are then put together by a so-called assembler into bigger fragments called contigs. Contigs are only the fragments of a genome, and the task of a metagenomic binner is to cluster all genome fragments that belongs to one genome together. In the context of metagenomic binning, the resulting clusters are called bins.

Most metagenomic binners do not utilize deep learning methods, although deep learning has seen a rise in popularity in recent years. Lately, deep learning methods have been applied to various domains, which often led to performance improvement in these areas [10]. General examples of areas non-related to metagenomic binning are language translation [11] and speech recognition [12]. Examples

of deep learning application in the area of metagenomics include metagenomic gene prediction [13] and metagenomic bacteria taxonomix classification [14].

Although deep learning is no stranger in the area of metagenomics, the application of deep learning for metagenomic binning is still in its beginnings. The best-known example of deep learning-based metagenomic binning is the variational autoencoder for metagenomic binning (VAMB) [1].

In a previous study, the group investigated the potential of VAMB [15]. A brief description of our findings can be found in the appendix (Appendix A.1). This study showed that abundance contributes little to the results.

1.1 Contribution

In this report, we want to explore which interesting possibilities deep learning bears for metagenomic binning by taking our previous findings into account [15]. We will provide an insight into two new areas that can be explored for metagenomic binning. Firstly, we will investigate the potential of using an alternative input feature. Secondly, we will study deep clustering methods suitable for metagenomic binning. Finally, we will compose new metagenomic binners, so that we can test and evaluate the mentioned ideas in practice.

Goal 1: Exploring Alternative Input Features Metagenomic binners take one or more input features that are provided as the input to the model. Those handcrafted input features are usually extracted from contigs. Handcrafted features are features that were manually designed from the original data, and created with the use of algorithms [16]. The most common features that are utilized by several binners, including VAMB [1] and MetaBAT2 [2] are composition and abundance.

- Composition (tetranucleotide frequency, TNF) refers to k-mer frequencies of a particular contig [17]. K-mers are substrings of length k calculated from the original sequence.
- Abundance refers to the similarity between aligned contigs.

An additional preprocessing step to obtain these input features is required before binning. Therefore, we want to investigate if it is possible to compose a model that can extract features directly from the contigs.

The first area that we want to investigate is the utilization of raw contigs as inputs of metagenomic binning model. In contrast to composition and abundance, contigs hold sequential information. That means that the order of nucleotides in a contig (sequence) is not random. By abstracting those sequences, it is possible

to identify features. Based on those learned features, contigs can be grouped into clusters, so-called bins.

The idea of extracting information from unprocessed DNA sequences using deep learning method has already been documented in scientific literature. Utilizing unprocessed DNA sequences as an input into a CNN-based model has showed promising results for metagenomic gene prediction [13]. Moreover, it has been already proven in a study, that it is possible to extract hierarchical features from genomic sequences [18].

Pattern recognition techniques, traditionally used for image recognition, might be a powerful tool for extracting features from sequences for the purpose of unsupervised clustering. In the current research, such techniques have been used for genomic classification tasks [13], [18]. Moreover, it is also used as a tool for extracting additional input features [19]. However, a metagenomic binner using unprocessed contigs as the only source of information have not yet been proposed in the scientific literature. By unprocessed contigs, we mean the original sequence of nucleotides that has been assembled from reads.

In this project, we will investigate and validate the viability of this idea.

Goal 2: Exploring Joint Deep Clustering Methods Binning refers to the process of clustering metagenomic contigs into the potential genomes [20]. The second area that we want to explore are deep clustering methods that could be promising for metagenomic binning.

Deep clustering methods usually involve feature representations learning, followed by clustering which is done on the learned representations. Feature representations, also known as embeddings, are machine-readable data that the deep learning network learns from the input data [21]. Those representations are the learned weights of the final embedding layer of the network [22].

There are multiple types of deep clustering. For instance, the deep learning-based binner VAMB is utilizing a so-called two-stage deep clustering method. Here, computing the feature representations and clustering forms two separable stages. The disadvantage of this approach is the risk of getting stuck in a local minimum because the learned representations might not be optimal for the clustering task [23].

Another approach is joint clustering, where the feature representations and clusters are learned simultaneously in an interconnected process. That way, clustering is involved deeper during the model training phase.

In this report, we will provide an insight into different deep clustering approaches. Specifically, we will put our focus on the area of joint clustering. Based on our findings, we will select a joint deep clustering approach that seems promising for metagenomic binning.

Proposed Solution In order to interconnect our two main goals, we will propose two metagenomic binners, that will let us explore and evaluate both of the defined areas. The performance of these binners will then be evaluated against state-of-the-art methods. Both of the binners will utilize the selected joint clustering technique. This will let us explore the benefits of joint clustering for metagenomic binning.

The difference between the two binners will be in the type of input features they are using. One of the binners will use raw contigs as inputs. The second binner will use composition and abundance input features.

Based on our goals, we propose the two following metagenomic binners:

- Deep Convolutional Metagenomic Binner (DCMB)

  The first binner combines our Goal 1 and Goal 2. It takes raw contigs as input, and it utilizes joint deep clustering technique.

- Deep Stacked Metagenomic Binner (DSMB)

  The second binner is related only to Goal 2. It takes composition and abundance as input, and it utilizes joint deep clustering technique.

1.2 Problem Statement

The existing metagenomic binners don't use unprocessed contigs as their input features. Instead, they use hand-crafted features calculated from these contigs. However, contigs contain sequential information that could potentially be used as an alternative source of information for a metagenomic binner. We assume that a sequential information-based binner could achieve competitive binning results compared to the state-of-the-art.
Moreover, to use the full potential of a deep learning-based metagenomic binner, the model must learn feature representations that will be optimal for the clustering task. Typically, deep clustering has two separable phases: feature representation learning and clustering. We will explore the options for interconnecting the two phases. We aim to explore, whether optimizing the learned feature representations and clustering simultaneously will lead to improved clustering, respectively binning results.
Therefore we investigate the following questions in this report:

1. Can we benefit from the sequential information in contigs and achieve competitive binning results by implementing a deep learning model that will use them as an input?

2. Can we benefit from involving clustering deeper during the training phase?

Chapter 2

Background

This chapter explores the methods, structures, and concepts crucial for deep learning and deep clustering in the area of metagenomic binner. In Section 2.2, the fundamentals of deep neural networks are explored, and different types of deep neural networks are examined.

The following section about the bioinformatics background is borrowed from our previous work [15].

2.1 Bioinformatics

Section 2.1.1 describes the characteristics and the general structure of DNA, the building block of all life, while Section 2.1.2 describes the field of metagenomics. In Section 2.1.3, the process of sequencing is explained based on the techniques used by the Illumina and Nanopore sequencers. Lastly, the metrics important for binning, abundance and composition are outlined in Section 2.1.4.

2.1.1 DNA

DNA (deoxyribonucleic acid) is a molecule composed of building blocks called nucleotides. These nucleotides form long chains which coil around each other into a structure called a double helix. Nucleotides are composed of four bases: Adenine, Cytosine, Guanine, and Thymine, along with two other groups, a deoxyribose group, and a phosphate group. The bases, normally represented with their initials, are the main carriers of information in DNA and always come in pairs: A-T and C-G. These base pairs then form larger groups called genes. Genes can have variable length and definition, but their main distinction is that they are the smallest unit of "useful" genetic information, i.e., information that instructs the body how to build proteins [24].

DNA has the unique property of being able to replicate itself by first unwinding its two complementary strands and then using each one as a template for replication. This process is also its main purpose. By doing this, DNA can pass on its information to the next generation of organisms, thus being preserved and growing in number. It can also be mutated by the change of a random base pair, thereby giving the organism it builds new properties and allowing it to adapt to its environment. Using this mechanism, DNA forms the basis of evolution and all life on Earth [25].

Long strands of DNA are bundled into packages called chromosomes. They are unique for every organism and come in pairs (e.g. humans have 23 pairs of chromosomes, rice has 12, dogs 39) [26]. The collection of all chromosomes in a cell is called a genome. It represents the entire genetic material of an organism and is present in every cell [27].

2.1.2 Metagenomics

Metagenomics is the study of environmental samples containing the DNA of multiple organisms. By treating the samples as one large genome (metagenome) we can identify the distinct species found within that sample, and possibly even discover new ones. This also poses a challenge, as we need to invent algorithms and methods that can process the vast amounts of data present in metagenomes and assign it to separate genomes based on the species. Metagenomics has a wide range of applications, ranging from the human gut microbiome to biofuel [28].

A very helpful tool for the identification of specific genomes within a metagenome are parts of DNA known as genetic markers. Genetic markers are DNA sequences whose position on a chromosome is known [29]. Because of this, a genetic marker can be used to identify species based on their genome. A special and important category of genetic markers for metagenomic binning are single-copy marker genes, as they have the property of occurring only once in a genome [30].

2.1.3 Sequencing

Sequencing is the process of identifying the sequence of nucleotide bases of a particular DNA molecule [31]. There are many different approaches for achieving this. This study focuses on the process used by the Illumina and Nanopore sequencers. We also describe the most common file types that are used in sequencing.

Nanopore Sequencing

The following description of the Nanopore sequencer is mainly based on [32]. The name comes from the nanopore, a hole in the size of nanometers. Attached to

the nanopore is a sensor that measures the ionic current of anything that passes through it.

In the first step, a DNA fragment is attached to the adapters of the Nanopore device. To allow contiguous sequencing of both DNA strands, they are covalently attached to each other. The DNA fragment is then passed through the Nanopore and a sensor captures changes in ionic current. Worthy of note is that the first strand that is read is called a “template read” and the complementary strand is called a “complement read”. In the next step, the changes are separated into distinct segments in regards to mean amplitude, variance, and duration. Every base has a different pattern of these characteristics, allowing us to translate these ionic changes into corresponding bases. This is then translated into *k-mers*. K-mers represent all possible substrings of length k in a given nucleotide sequence [33], with k here ranging from 3-6. An example of all 4-mers of the sequence "ATGCATAT" is shown in Figure 2.1.

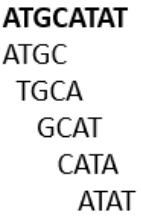


Figure 2.1: All possible 4-mers for the sequence "ATGCATAT"

Illumina Sequencing

The process of sequencing used by Illumina is described based on [34] and visualized in Figure 2.2. In the first step, the DNA sample is randomly split into smaller sequences. Next, complementary adapters are attached to both ends of a sequence as a marker and the marked sequences are put on a solid plate called a flow cell (A). In the next step, several copies of each sequence (clusters) are created by using a process called PCR bridge amplification (B). The last step is to determine the nucleotide order of the sequences (C). First, a *primer* is attached to the adapter on the top of the sequence (marked here in green). Then, the complementary nucleotide for the first nucleotide in the sequence is attached to the primer (marked in red and yellow). These complementary nucleotides are fluorescently labeled, where each of the 4 DNA bases has a different fluorescent label. Next, a laser scans the cluster to determine the fluorescence, and therefore the added base. This process is then repeated for the entire sequence until the bottom adapter (purple) is reached.

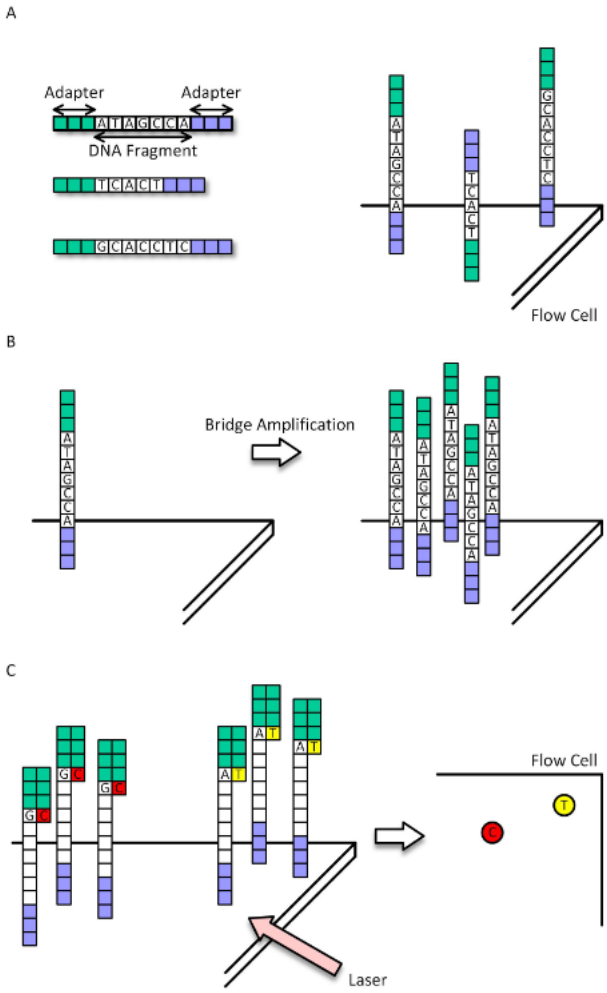


Figure 2.2: Illustration of the three stages (A, B and C) of Illumina sequencing[35, p. 5].

Comparison

For all DNA sequencing applications besides De novo sequencing, Illumina has a recommended read length of 150 bp (base pairs) [36], while the MinION device is advertised of having an unrestricted read length [37]. Practical advantages of the Nanopore technology are that the devices are portable and start from 1000$. On the other hand, a disadvantage of the Nanopore technology is that the read accuracy is lower than the 99% accuracy achievable with short-read technologies like Illumina [38]. A comparison of both devices on sequencing Campylobacter jejuni strains showed that the MinION called 1 in 10 bases incorrectly, while the Illumina miscalled 1 in 36 bases [39].

Assembly

After sequencing is done, we are left with many fragmented short reads that do not provide sufficient information about the original genome. Due to this, we need assemblers to combine these short reads into longer ones. However, this task is challenging for many reasons, some of them being read errors, repetition of small sequences within the original sequence, miss-assembled sequences, and many others. Tools for dealing with this task are e.g. the Burrows-Wheeler Aligner [40] for short read alignment and assembling, or the newer Minimap2 [41]. The results of the assembly step can be represented as files with the *.fasta* or *.contigs* extensions.

Other Common Preprocessing Data Structures

Sequence Alignment/Mapping - SAM Files with the *.sam* extension contain biological sequence data in the tab-separated format. SAM files contain an optional header and an alignment section. The header, when present, must be located prior to the alignment section and has to start with an "@" character. Alignment, however, starts without "@" and contains 11 mandatory fields for important alignment information among other optional fields for additional data coming from the alignment.

Binary sequence Alignment/Mapping - BAM Files with the extension *.bam* are essentially SAM files in a binary format for space saving. They are used as the main input for many binners. In practice, a BAM file's contents can be sorted by many criteria, some of them being alignment length, read name, etc. A sorted BAM file can sometimes speed up the binning process.

2.1.4 Abundance and Composition

The composition of a sequence is a metric that provides information about the structure of a sequence at the k-mer level. Composition describes with which percentage a k-mer occurs in a contig. It is represented as a vector that stores the number of occurrences of each k-mer in the sequence. Due to the need of comparing different sequences, the composition vector is normalized over the sum of all counts, e.g. a normalized composition vector for AATGCAAT for the k-mers of size 3 would be the following:

3-Mer	Probability
AAT	2/6
ATG	1/6
TGC	1/6
GCA	1/6
CAA	1/6

Table 2.1: Composition example for the sequence "AATGCAAT".

We use the composition metric due to the fact that the composition of closely related genomes is usually very similar. Most often, composition is created using 4-mers, as there is a suitable number of 4-mers. The vector of 4-mers has a size of 136, much more than a vector of 3-mers and much less than a vector of 5-mers, giving it a balance between expressiveness and computational cost. We have to note that 136 is fewer than the number of 4-mers. This is the case because we have redundant information when using all 4-mers, as there are 120 reverse complement 4-mers like GTAA - TTAC and 16 palindromes like TAAT [42, p. 5].

The other metric we use, which is the abundance or coverage of a sequence, provides information about how similar the sequences of several aligned reads are. Sequences with similar abundance are likely to be from the same genome. There exist several approaches for calculating abundance. For instance, VAMB uses reads per kilobase per million mapped reads (RPKM) as its abundance value [43, p. 2]. The RPKM score is calculated according to the following formula [44]:

$$RPKM = 10^9 \times \frac{Reads\ mapped\ to\ the\ transcript}{Total\ reads\ \times Transcript\ length}$$

RPKM is used to essentially rescale the gene counts using the length of the gene and the number of mapped reads.

2.2 Deep Neural Networks

Deep learning is an area of machine learning that utilizes deep neural networks for problem-solving [45]. The concept of deep neural networks is inspired by the neurons of the human brain [46]. The basic unit of a deep neural network is the perceptron. The perceptron takes as input a n-dimensional input vector $\mathbf{x}$ with $(x_1, ..., x_n)$. Connected is a n-dimensional weight vector $\mathbf{w}$ with $(w_1, ..., w_n)$. The perceptron computes a single output y by the following formula [47]:

$$\mathbf{x} \cdot \mathbf{w} = y \tag{2.1}$$

Commonly, a bias weight denoted with b is added to the formula to fit the network better to the data. The formula to calculate the output, including the bias, is defined

as the following:

$$\mathbf{x} \cdot \mathbf{w} + \mathbf{b} = y \tag{2.2}$$

The bias can also be included in the weights. In that case a constant of 1 must be added to the input.

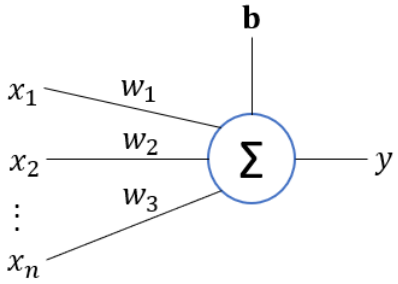


Figure 2.3: Illustration of the perceptron. Inspired by [48].

To transform the output of a perceptron, the output can be passed through an activation function ϕ for non-linear transformation. The most common activation functions are:

- Sigmoid function
- Tanh function
- ReLU function
- Softmax function

Sigmoid function

The sigmoid function is defined by the following equation[47] and visualised in Figure 2.4:

$$sig(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

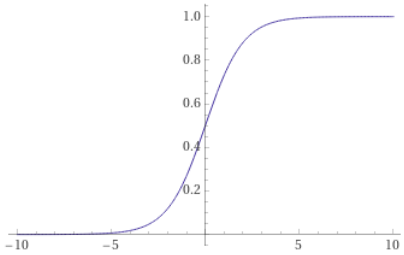


Figure 2.4: Visualisation of the sigmoid function [49].

The sigmoid function is differentiable and monotonic. As the range of the function is from 0 to 1, the sigmoid function is well suited for problems that involve probabilities.

Tanh function

The tahn function is defined by equation[46]:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

The tanh function is visualised in Figure 2.5:

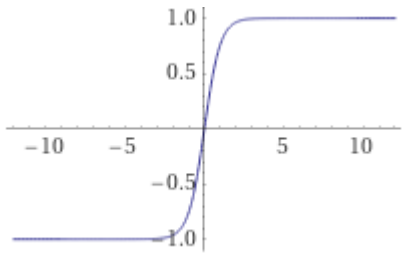


Figure 2.5: Visulisation of the tanh function.

The tanh function has a range from -1 to 1 and its shape is similar to the sigmoid function. Additionally, the tanh function is also differentiable and monotonic. Due to its range, the tanh function is often used for classification problems.

ReLU Function

The ReLU (Rectified Linear Unit) function is defined by the following formula [47]:

$$ReLU(x) = max(0, x) \tag{2.5}$$

The function is illustrated in Figure 2.6.

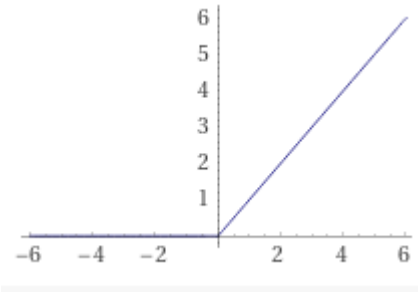


Figure 2.6: Visualisation of the ReLU function.

The ReLU function outputs 0 if the input is ≤ 0 else the output value is equal to the input value. The ReLU function is only non-differentiable at zero and monotonic.

Softmax Function

The softmax activation function probabilistically expresses the likelihood of the predicted values in a vector by scaling them, so that these probabilities sum up to 1 [50]. The softmax function is defined as:

$$softmax(\vec{x})_i = \frac{e^{x_i}}{\sum_{n=1}^{N} e^{x_n}} \tag{2.6}$$

Here, $(\vec{x})_i$ is the i-th input vector, and e^{x_i} is a standard exponential function applied to each element in the vector. The denominator normalizes the output values.

2.2.1 Neural Networks

Neural networks are created by organising different neurons into layers. A basic neural network is illustrated in Figure 2.7.

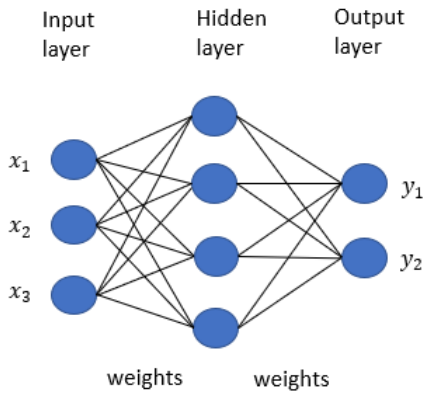


Figure 2.7: Basic neural network structure. Inspired by [51].

The first layer of a neural network is called the input layer, and the last layer is called the output layer. The input layer has the same size as the input vector x. The middle layers are also known as hidden layers. In Figure 2.7, the network consists of 3 input neurons, 4 neurons in the hidden layer and 2 neurons in the output layer. The lines between the layers are representing the weights. In a feedforward

network, each neuron performs the following equation to compute an output y:

$$y = \phi(\mathbf{b} + \sum_{i=1}^{n} x_i w_i) \tag{2.7}$$

To note is that the output of the hidden layer is denoted with h.

In Equation (2.7), ϕ is an activation function, $\mathbf{b}$ the bias, x_i is the i-th input and w_i is the weight of the i-th element.

The loss function is an essential part of training a neural network. It evaluates to which degree can a network output the correct value for a given input, and it compares the output with the correct value and indicates how well the network performed. While training, the neural network tries to minimize the loss function. There exists a wide range of loss functions where each one is suitable for a certain problem that should be solved. Two commonly used loss functions are the cross-entropy loss and the mean squared error loss. As we will see later in this report, sometimes searching for the optimal solution involves working with special and less frequently used loss functions such as the cosine similarity loss.

There are several variations of the cross-entropy loss. The most common two are the categorical cross-entropy loss and the binary cross-entropy loss. The categorical cross-entropy loss is used primarily for multi-class classification tasks [52]. It assumes, that a single class out of several possible ones is correct. Since the target labels are one-hot encoded, in situation where there are 4 possible categories and first one is labeled as correct, the output vector has a shape [1,0,0,0].

$$L_{CE} = -\sum_{i=1}^{x} y_i log(p(y_i)) \tag{2.8}$$

In Equation (2.8), y_i is the truth label, $p(y_i)$ is the softmax probability of the predicted label belonging into i-th class and x is the number of classes.

The binary cross-entropy function is used for models that output a probability between 0 and 1, and for binary classification tasks. It differs from categorical cross-entropy loss by considering each output individually [52]. It is defined by the following formula:

$$L_{BCE} = -\sum_{i=1}^{x=2} y_i log(p(y_i)) + (1 - y_i) log(1 - p(y_i)) \tag{2.9}$$

In Equation (2.9), y_i is the correct label and $p(y_i)$ is the predicted probability. The number of classes x is set to 2, because the truth values are either 0 or 1.

Mean squared error is used for models that output a continuous real number and it is defined by:

$$L_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - p(y_i))^2 \tag{2.10}$$

In Equation (2.10), y_i is the ground truth label of the i-th data-point, and $p(y_i)$ is the predicted label for the i-th data-point.

Cosine similarity loss is measuring the cosine of the angle between an input vector and a predicted vector. The higher the similarity between the vector is, the higher is the accuracy of the prediction. The loss outputs values in between -1 (perfectly opposite vectors) and 1 (identical vectors) [53]. The loss is expressed as:

$$L_{COS} = cos(\theta) = \frac{y \cdot p(y)}{\|y\|\|p(y)\|} = \frac{\sum_{i=1}^{N} y_i p(y_i)}{\sqrt{\sum_{i=1}^{N} y_i^2}\sqrt{\sum_{i=1}^{N} p(y_i)^2}} \tag{2.11}$$

Usually, the cosine similarity loss is used to compare the similarity between two documents [53]. However, the loss is also useful when comparing similarity between two mathematical vectors.

Gradient descent is an optimisation algorithm that will find the minimum of a function. In this setup, the goal is to find the set of weights for the lowest loss. It utilizes the gradient to a function to push the weights towards the minimum of the function. The gradient indicates the opposite direction of the minimum. Therefore, the weights can be adjusted by updating them towards the negative of the gradient. It is defined by following formula:

$$w = w - \gamma \cdot \nabla f(w) \tag{2.12}$$

w represents the weights of the function and $\nabla f(w)$ the gradient of the function. Here, γ is the so-called learning rate. The learning rate is a factor multiplied by the gradient in order to control how big the steps that are taken towards the minimum are. A small learning rate increases the optimization time. A too-large learning rate can lead to missing the minimum.

The pseudocode for this algorithm is provided below [46].

Algorithm 1 Gradient Descent Algorithm

```
Initialize weights randomly
repeat
    Compute gradients ∇f(w)
    Update weight w ← w − γ · ∇f(w)
until convergence of weights
return weights
```

Firstly, the weights w are randomly initialised. Next, the gradients of the prediction function f with the specific values for w are computed. This process is repeated until convergence.

2.2.2 Backpropagation

This subsection is a slightly modified version of the subsection with the same name from our previous report [15].
Backpropagation is an algorithm used for computing the gradients of a loss function in order to determine how training examples should optimize the weights **w** and biases **b** of a neural network. It does this in two ways:

1. It determines whether they should increase or decrease
2. It calculates the proportions of those changes in a way that optimizes the loss function in the fastest possible way

A true gradient descent step would involve doing this for all the training examples in our dataset and then calculating the average value. The process is computationally intensive, so alternatively, data can be subdivided into minibatches with a smaller number of examples. The other extreme would be to use just one training example, but this turns out to be worse than the minibatch version due to being less precise.

The Backpropagation Algorithm

In general, there are three requirements for the backpropagation algorithm:

- **A dataset** with input-output pairs (x_i, y_i), where x_i is the input vector and y_i is the output vector. Given N is the size of input-output pairs, the dataset can be represented as $X = \{(x_1, y_1), ..., (x_N, y_N)\}$
- **A feedforward neural network** where there are no connections between nodes within the same layer, while being fully connected between adjacent layers.
- **An error function** $E(X, \theta)$ that defines the error between correct label y_i and the predicted output $p(y_i)$. The gradients of this error function are updated in each iteration t during the training of the neural network. An iteration of the algorithm can be described by the following equation, where γ denotes the learning rate and θ denotes the parameters of the network, i.e. its weights **w** and biases **b**:

$$\theta^{t+1} = \theta^t - \gamma \frac{\partial E(X, \theta^t)}{\partial \theta}$$

Generic Backpropagation

There are five equations needed in order to understand the generic backpropagation algorithm:

- Calculate partial derivatives:

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} \tag{2.13}$$

  where $\frac{\partial E}{\partial w_{ij}^k}$ is the partial derivative of the error function E in respect to weights w_{ij}^k. It equals to the error term δ_j^k (the error at node j in layer k) multiplied by o_i^{k-1}: the output of node i in layer $k-1$.

- Calculate error terms for the output layer:

$$\delta_1^m = \phi_o'(a_1^m)(p(y) - y) \tag{2.14}$$

  where ϕ_o' is the derivative of the output layer's activation function, m is the index of the final layer, a_1^m is the product sum plus bias for node 1 in the m-th layer.

- Calculate error terms for the hidden layers:

$$\delta_j^k = \phi'(a_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \tag{2.15}$$

  where ϕ' is the derivative of the hidden layer's activation function, a_j^k is the product sum plus bias for node j in layer k, l counts all layers from index 1 to r^{k+1} (the number of nodes in the next layer)

- Aggregate partial derivatives of each input-output pair:

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^j} = \frac{1}{N} \sum_1^N \frac{\partial}{\partial w_{ij}^k} (\frac{1}{2}(p(y_d) - y_d)^2) = \frac{1}{N} \sum_1^N \frac{\partial E_d}{\partial w_{ij}^k} \tag{2.16}$$

- Update the weights:

$$\Delta w_{ij}^k = -\gamma \frac{\delta E(X,\theta)}{\delta w_{ij}^k} \tag{2.17}$$

After selecting a suitable learning rate γ and randomly initializing parameters w, the algorithm will go through the following steps:

1. **Calculate the forward pass** of each input-output pair (x_d, y_d) and **store** a set of three results $p(y_d), a_j^k, o_j^k$ for each node j in the layer k by processing values from the input layer 1 to the output layer m.

2. **Calculate the backward pass** of each input-output pair (x_d, y_d) and **store** the results of $\frac{\partial E_d}{\partial E_{ij}^k}$ for each weight w_{ij}^j which is connecting node i of layer $k-1$ to node j in layer k by processing all values from output layer m to the input layer 1. This is done in the following 3 steps:

   (a) Use equation (2.14) to evaluate the output layer's error term

   (b) Go backwards through the hidden layers δ_j^k and use equation (2.15) to propagate the error terms from the last hidden layer $k = m - 1$

   (c) Use equation (2.13) to calculate partial derivatives.

3. **Aggregate gradients** for each input-output pair $\frac{\partial E_d}{\partial E_{ij}^k}$ to get the final gradient $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$ for all input-output pairs $X = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ by using equation (2.16).

4. **Update the weights** using the learning rate γ and the sum of gradients $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$ using equation (2.17).

2.3 CNN

The CNN (Convolutional Neural Network) is a type of ANN (Artificial Neural Network) that extracts higher representations of data. It consists of a set of feed-forward layers. CNN is mostly used for classification tasks on image data, but can also be used for data analysis [54]. The hidden layers of CNN are called convolutional layers. The two main operations within a convolutional layer are convolution and pooling [46].

2.3.1 Convolution

Convolution is an operation that abstracts high-level features from the input. The input of each convolutional layer is a multidimensional array. This multidimensional input is referred to as a *tensor*. The input itself is called a channel. The weight tensor of a CNN is called a filter or *kernel*. Typically, the kernel has smaller size than the channel.

An example of a one-dimensional kernel is $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$.

With the kernel, it is possible to abstract features from the input by performing convolution. Defined is the convolutional operation as [55]:

$$(f * g)(i) = \sum_{j}^{m} g(j) \cdot f(i - j + m/2) \tag{2.18}$$

Here, f is the input tensor, g is the kernel, and n and m are the lengths of the input tensor, respectively, the kernel. The output is referred to as the *feature map*. A convolution can basically be viewed as a dot product between an input and a kernel, with the goal to obtain a feature map. The values of the kernel and the input channel are multiplied and summed up to compute each value of the feature map. The feature map has reduced dimensionality compared to the input. An important parameter in the context of convolution is the stride. Stride defines, if the convolution should be evaluated for every column and row or only for every n-th column and row. It is also possible to have multiple input channels, where *depth* refers to the number of input channels.

An example of convolution is given in Figure 2.8.

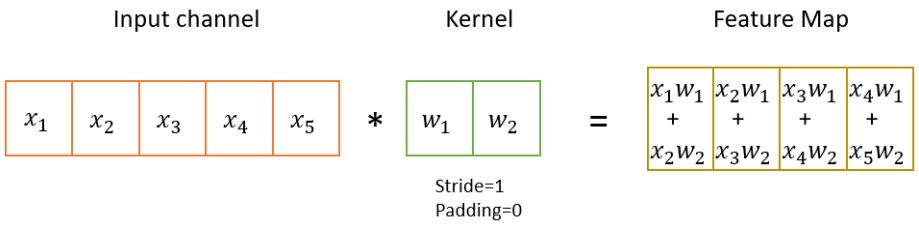


Figure 2.8: Convolution example. Here, the input matrix has a size of 5, and the kernel has a size equal to 2 units. The stride is set to 1, and the feature map is of a size 4 [56].

A CNN has a sparse connectivity, because it detects meaningful features using a kernel that is smaller than the input. Another characteristic of CNN is that the weight tensor (kernel) is used at every position of the weight. This contrasts with an ANN such as autoencoder where the weights are not shared.

Through performing convolution, the output will be shrunk to smaller dimensions. This shrinking can be a problem for the following layers if the size of the output gets very small. Another problem is that information from the edges is lost, as the filter will take the other areas more into account due to the kernel slides. A solution to this is to pad the input tensor around the tensor border. Commonly, the tensor is padded with values of 0.

Here, p is the padding amount. An example of padding is given in Figure 2.9.

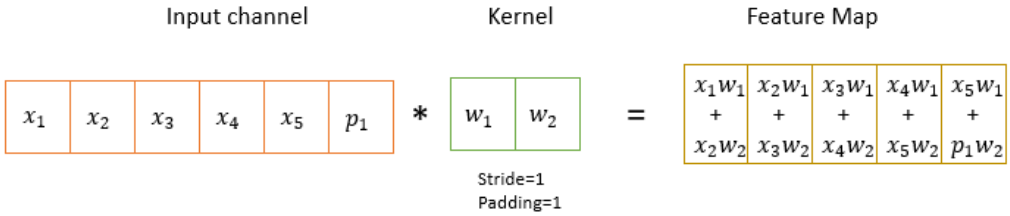


Figure 2.9: Padding example with padding set to 1 [57].

After the convolution, a non-linear activation function is applied.

2.3.2 Pooling

A pooling layer reduces the size of the representation and achieves translational invariance. Translational invariance means that the location of a feature will not be taken into account [58]. Each pooling layer performs a pooling operation. The most common pooling operations are:

- Max Pooling
- Average Pooling

Max Pooling

Instead of computing a dot-product between input and kernel, a feature map value is computed by taking the maximum value of the region. Max pooling abstracts the most prominent features [58] of each input region. An example of max pooling is illustrated in Figure 2.10.

Figure 2.10: Example of max pooling. In the example, the kernel has a size of 2 units and the pooling size is 2. For each of the coloured areas, the maximum value of the area is abstracted.

Average Pooling

Average pooling abstracts the average features [58] of each input region. Figure 2.11 visualizes the example of this operation.

Figure 2.11: Example of average pooling. In the example, the kernel has a size of 2 units and the polling size is 2. For each of the coloured areas, the average value of the area is abstracted.

2.4 Autoencoder

This section is a slightly modified version of the subsection with the same name from our previous report[15].

An autoencoder is an unsupervised artificial neural network that allows for the encoding and subsequent decoding of data. In order to perform these operations, an autoencoder is split into two parts [59]:

- An encoder $f : R^n \rightarrow R^p$
- A decoder $g : R^p \rightarrow R^n$

The encoder is tasked with reducing high-dimensional input data of dimension n into a lower-dimensional embedding of dimension p. The decoder, on the other hand, tries to recreate the input data from these embeddings. In other words, a lower-dimensional embedding of dimension p is transformed into a high-dimensional output of dimension n. This way, the autoencoder is forced to learn the characteristics of the input data, instead of just storing all of the input data's information inside of its nodes. The functions f and g that define the encoder and decoder are trained by minimising a loss function L in the following way [58]:

$$L(x, g(f(x)))$$

This loss function is called a reconstruction loss and it is tasked with penalizing the dissimilarity between the input x and the decoded encodings of x. The most commonly used loss functions are the mean squared error and the cross-entropy loss.

Both the encoder and decoder are consisting of layers of interconnected neurons, not too dissimilar from a standard neural network. They consist of an input layer and an output layer, in addition to one or more hidden layers. Figure 2.12 illustrates this structure.

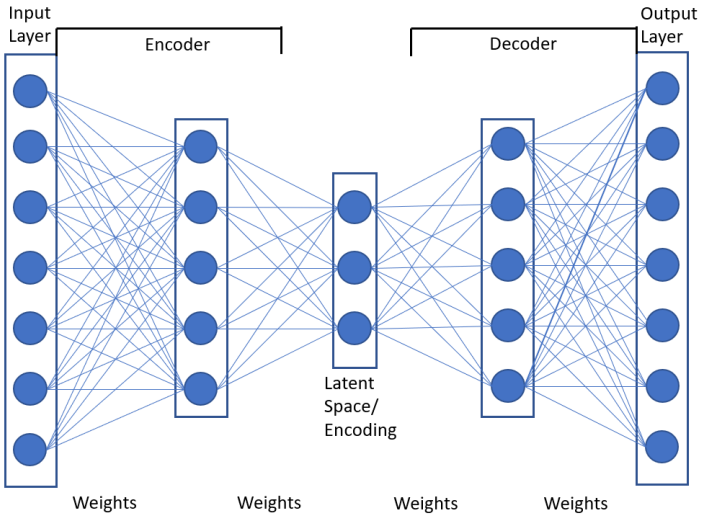


Figure 2.12: Illustration of the basic autoencoder structure. Inspired by [60].

2.4.1 Stacked Autoencoder

A stacked autoencoder is a variant of an autoencoder that consists of more than two autoencoders stacked together. The autoencoders are stacked together layer by layer, meaning that the first layer is pretrained before the next one. The output of the hidden layer is the input of the next hidden layer. Layer-wise pretraining means that the output of one layer is the input for the next one. Each layer of the staged autoencoder is trained by minimizing the reconstruction loss in this pretraining. After pretraining, back-propagation is used to fine-tune the autoencoder. [61]. The general structure of a stacked autoencoder is visualized in Figure 2.13.

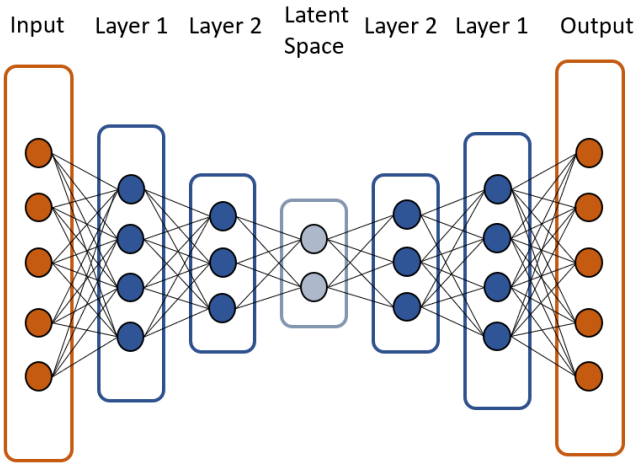


Figure 2.13: Example for the general structure of a stacked autoencoder. Inspired by [62].

2.4.2 Variational Autoencoder

This section is a slightly modified version of the subsection with the same name from our previous report[15].

A variational autoencoder is a generative probabilistic model specialised for learning latent representations [63], standing in contrast to a generic autoencoder. A variational autoencoder learns these latent representations by modeling a distribution, often a normal distribution, over the input data. It can then reconstruct instances similar to the original data from this distribution with some variation added to it. The structure of a variational autoencoder is shown in Figure 2.14.

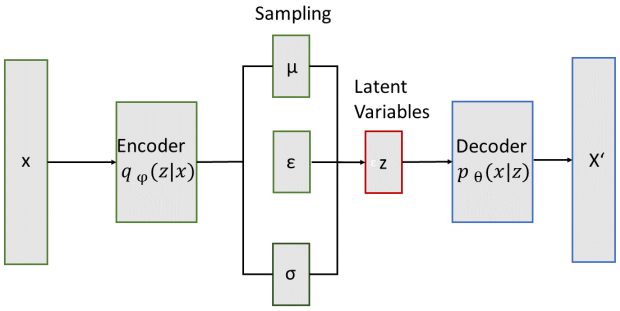


Figure 2.14: Structure of a variational autoencoder. Input data x is encoded using the encoder $q_\phi(z|x)$ into a latent space z via a normal distribution with mean value μ and covariance σ. When reconstructing the data using the decoder $p_\theta(x|z)$, the data is sampled from the normal distribution with added randomness ϵ. Inspired by [63]

A variational autoencoder has to tackle the problem of the calculation of the true posterior $p_\theta(z|x)$ of a distribution being intractable. To solve this, the true posterior distribution can be approximated with a recognition model $q_\phi(z|x)$. More precisely, let us assume that we have a dataset $X = \{x^i\}_{i=1}^N$ that consists of N samples of a variable x. We are now faced with the problem that the process by which the data was generated has some latent variable z. z is generated from a prior distribution $p_\theta(z)$ and x with the likelihood $p_\theta(x|z)$ [64]. The parameter θ and the values of the latent variable z are unknown to us.

To solve that problem, the generative model tries to approximate the true posterior distribution with a multivariate Gaussian. The reason for this is that the Gaussian distribution is easy to model due to having only two parameters. The effect of these parameters on the distribution is shown in Figure 2.15.

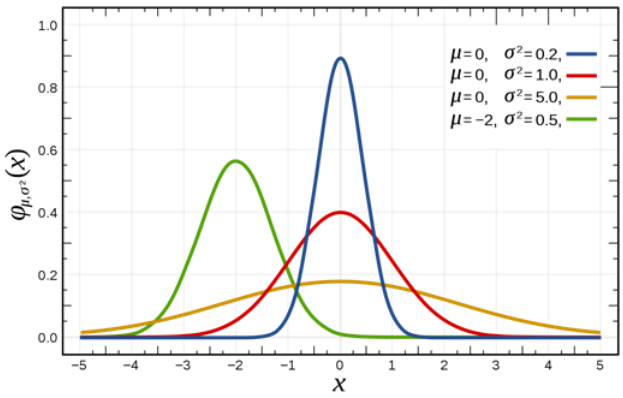


Figure 2.15: Illustration of the Gaussian distribution for different values of μ (mean) and σ^2 (variance) [65].

To optimize the distribution, it is necessary to learn the parameters θ and ϕ. In the process of learning, we will use a similarity measure called Kullback–Leibler divergence (D_{KL}). The D_{KL} measures how dissimilar two distributions p and q are from each other [58]. It is defined using the following equation:

$$D_{KL}(q||p) = \sum_{\{x\} \in X} q(x) \log \frac{q(x)}{p(x)}$$

with the properties:

- $D_{KL}(q||p) = 0$ iff $p = q$
- $D_{KL}(q||p) \geq 0$ else

This means that D_{KL} is always greater or equal to 0 whenever the distribution q is not similar to the distribution p. The greater the value of $D_{KL}(q||p)$ is, the

more dissimilar the distributions q and p are. We also need to note that $D_{KL}(q||p)$ is asymmetric in regards to q and p, meaning that $D_{KL}(q||p) \neq D_{KL}(p||q)$. In the context of deep learning, the D_{KL} is often use to match an auxiliary q distribution to a target distribution p. To match the distributions, the $D_{KL}(q||p)$ is minimised.

We can rewrite the D_{KL} if we take the following relation with marginal likelihood into account. The marginal likelihood $\log p_\theta(x^{(1)}, ..., x^{(N)})$ can be generally written as the sum over the marginal likelihoods of the data points in the dataset $\sum_{i=1}^{N} \log p_\theta(x^{(i)})$. The marginal likelihood $\log p_\theta(x^{(i)})$ can now be rewritten as [64]:

$$D_{KL}(q_\theta(z|x^{(i)})||p_\theta(z|x^{(i)})) + L(\theta, \phi; x^{(i)})$$

This expression consists of two terms: the D_{KL} and the ELBO (Evidence Lower Bound), also known as the variational lower bound. The ELBO represents a lower bound to $\log p_\theta(x^{(i)})$. Minimizing the D_{KL} is equivalent to maximizing the ELBO. Therefore, we can optimize the parameters θ and ϕ by maximising the ELBO using stochastic gradient ascent during training. The ELBO itself can be rewritten to:

$$L(\theta, \phi; x^{(i)}) = -D_{KL}(q_\phi(z, x^{(i)})||p_{\theta(z)}) + E_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)}|z)]$$

During optimization, the D_{KL} term in this equation has the purpose of regularizing the parameters, while the second term represents the expected negative reconstruction error. However, before learning can be performed, we need to reparameterize the encoder. The reason for this is illustrated in Figure 2.16.

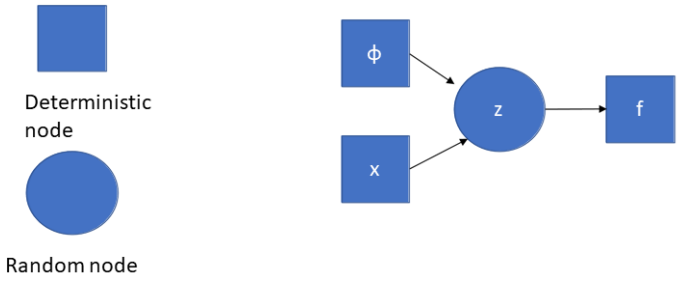


Figure 2.16: Illustration of the parameterization problem for the encoder. Inspired by [66].

In this figure, the variable z is a random variable. The problem we encounter here is that backpropagation is not suited for passing through a random node. Therefore, we need to outsource the property of randomness to a different node ϵ. We do the following. First, we redefine z as a deterministic variable $z = g_\phi(\epsilon, x)$. ϵ is a component of added noise, and it is defined with a distribution $p(\epsilon)$. $g_\phi(\cdot)$ is a vector valued function, meaning that it takes a value as input and returns a vector. Hence, we can reparameterize the model by adding the noise ϵ as a randomness component, as illustrated by Figure 2.17.

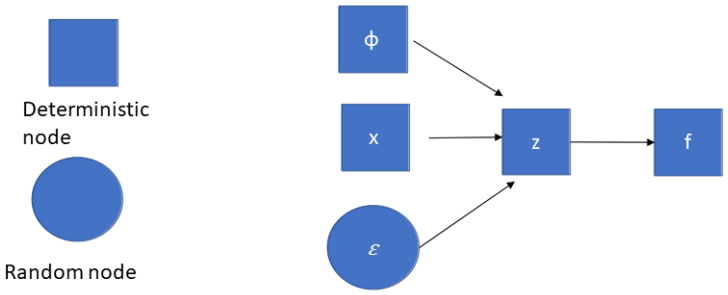


Figure 2.17: Illustration of the reparameterization trick. Inspired by [66].

For the variational autoencoder, we parameterize $p(z)$ with the normal distribution $p_\theta(z) = N(z;0,1)$. Furthermore, the recognition model is parameterized by $q_\phi(z|x) = N(z;\mu,\sigma^2)$. With that being done, it is now possible to use the SGVB (Stochastic Gradient Variational Bayes) estimator to learn the parameters θ and ϕ. The ELBO of this estimator is defined as [64]:

$$L^A(\theta,\phi;x^{(i)}) = \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(x^{(i)},z^{(i,l)}) - \log q_\phi(z^{(i,l)}|x^{(i)})$$

where

$$z^{(i,l)} = g_\phi(\epsilon^{(i,l)},x^{(i)})\epsilon^{(l)} \sim p(\epsilon)$$

For training, the AEVB (Auto-Encoding Variational Bayes) algorithm is used in a minibatch version. The algorithm is visualized in Algorithm Algorithm 2.

Algorithm 2 Minibatch version of the Auto-Encoding Variational Bayes (AEVB) algorithm.

$\theta,\phi \leftarrow$ Initialize parameters
repeat
  $X^M \leftarrow$ Random minibatch of M data points (drawn from full dataset)
  $\epsilon \leftarrow$ Random samples from noise distribution $p(\epsilon)$
  $g \leftarrow \nabla_{\theta,\phi}\widetilde{\mathcal{L}}^M(\theta,\phi;X^M,\epsilon)$ (Gradients of minibatch estimator)
  $\theta,\phi \leftarrow$ Update parameters using gradients g (e. g. SGD or Adagrad)
until convergence of parameters (θ,ϕ)
return θ,ϕ

In each step, a random minibatch of size M is drawn from the dataset together with samples from a noise distribution. Next, the gradient of the ELBO in regards to the current values of θ, ϕ, and the samples is calculated. The parameters θ and ϕ are updated in accordance with the gradient, and the process is repeated until the parameters converge or we reach the last iteration.

2.4.3 Convolutional Autoencoder

The CAE (convolutional autoencoder) is an ANN structure that applies the CNN on an autoencoder framework. The characteristic of a CAE is that the autoencoder is built with convolutional layers. Defined is the CAE as the following [67]:

$$e_W(x) = \sigma(X * W) = z \tag{2.19}$$

$$d_U(z) = \sigma(z * U) \tag{2.20}$$

Here, W and U are tensors, and $*$ refers to the convolutional operator. Like the plain autoencoder, the encoder transforms the input x into a latent representation z. The decoder restores the original input. CAEs are used for example to detect anomalies [68] or for feature extraction [69].

2.5 Clustering Techniques

In this section we explore clustering techniques that are frequently used by deep clustering methods.

2.5.1 K-means Clustering

K-means is an iterative clustering algorithm that groups n data points into k clusters. Given a dataset $X = \{x_1, ..., x_n\}$, the k-means algorithms forms k clusters, by assigning each datapoint x_i with i from 1 to n to one of the k clusters. The number of clusters k is a hyperparameter. K-means is an iterative algorithm that refines the cluster assignment for each iteration of the algorithm. For each cluster k there exists a cluster centroid μ_j with j in $\{1,..,k\}$. The centroids are also referred to as cluster centers. At the start of the algorithm, the centroids are assigned to random data points or random points in the cluster space. In each iteration, every point x_i is assigned to the closest centroid and the centroids are refined. The distance from the data point to the centroid is measured by a distance function such as the Euclidean distance. K-means use the following loss function to minimize the total mean squared error between data points and centroids [70]:

$$L_{KM} = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i - \mu_j||_2^2 \tag{2.21}$$

The loss function describes that every data point gets assigned to the centroid with the minimum distance between the centroid and data point. After assigning each data point to the current closest centroid, the position of each centroid gets

updated. The new position for each centroid μ_j is calculated by the following formula: [70]:

$$\mu_j = (\frac{1}{m_j}) \sum_{i=1}^{m_j} x_i \tag{2.22}$$

In this equation, m_j represents the number of data points of the j-th cluster. The equation shows that the centroids are updated according to their mean data point. The assignment of each data point to a cluster and the update of the centroid is done iteratively until the cluster assignments converge. K-means will try to cluster the data points into a circular form, which makes this method unsuitable for certain datasets. Additionally, the quality of the cluster assignment is highly determined by the initial centroid assignments.

2.5.2 KL Divergence Clustering

Another centroid-based method is the KL divergence clustering. It is a soft assignment clustering method, meaning that every data point is assigned to each cluster with a certain probability [70]. First, initial centroids μ_j with j from 1 to the number of clusters k are obtained by running k-means. For refinement of the centroids μ_j, the algorithm is minimizing KL divergence. KL divergence is a measure of a difference between two distributions **[kl]**, in this case an auxiliary target distribution P and a soft-assignment distribution Q. The KL divergence loss function is utilized to minimize the distance between the two distributions and it is defined as follows:

$$L_{KLD} = KL(P||Q) \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}}) \tag{2.23}$$

Soft assignments are the current cluster predictions. Target distribution represents the ideal clusters, and it is up to the deep learning method that uses this technique to define such ideal clusters. Target distribution takes into account high confidence data points.

In each iteration of the algorithm, the soft assignment probability q_{ij} and the target distribution p_{ij} are computed for each data point i and each cluster j. Next, each cluster centroid μ_j is updated according to the gradients of the KL divergence loss and the previous centroids. This process is repeated iteratively until convergence, or until the maximum iteration is reached. The soft assignments for each data point i and cluster j are computed by the following formula:

$$q_{ij} = \frac{1 + \|x_i - \mu_j\|^2)^{-1}}{\sum_j (1 + \|x_i - \mu_j\|^2)^{-1}} \tag{2.24}$$

The target distribution to obtain the high confidence data points is calculated. It is obtained by computing the soft cluster frequencies and defined by the following

formula:

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_j (q_{ij}^2 / \sum_i q_{ij})} \tag{2.25}$$

Lastly, each cluster centroid μ_j is updated by the following formula:

$$\mu_j = \mu_j - \frac{\lambda}{n} \sum_{i=1}^{n} \frac{\partial L_{KLD}}{\partial \mu_j} \tag{2.26}$$

In this equation, ∂L_{KLD} and $\partial \mu_j$ are the gradients of the KL divergence loss and the centroid.

2.6 Deep Clustering

Architectures like the autoencoder learn lower dimensional data representations from the input. Clustering on these deep representations or embeddings obtained from a deep learning method is called deep clustering. Clustering on these embeddings can improve the clustering results and the cluster extraction. The learning of embeddings can be seen as a dimensionality reduction in enabling clustering on lower-dimensional data. Deep clustering approaches vary on the used architecture, loss functions, and algorithmic. Nevertheless, deep clustering approaches can be sorted according to the way they structure representation learning and clustering into the following three categories[71]:

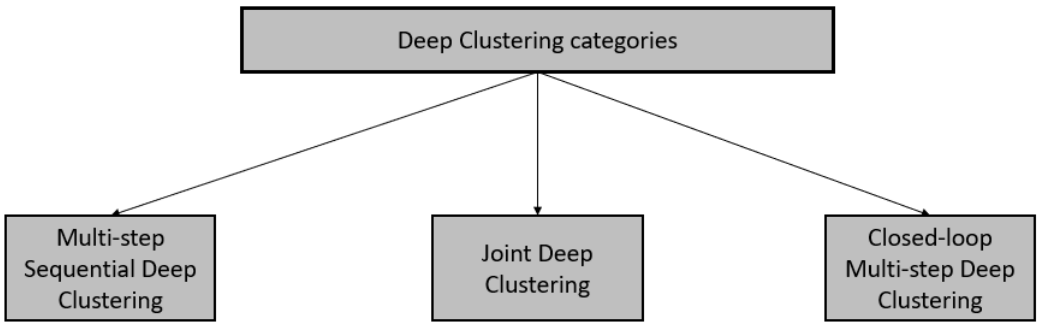


Figure 2.18: The 3 categories of deep clustering. Inspired by [71].

The multi-step sequential deep clustering approach is split up into two stages. In the first stage, a deep representation is obtained from the input. This representation is then utilized in the second step for clustering.

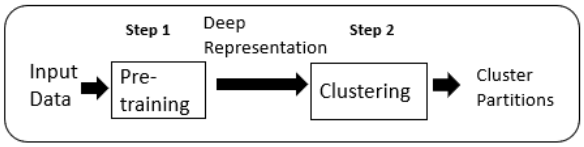


Figure 2.19: Multi-step deep clustering approach structure. Inspired by [71].

The joint deep clustering approach combines representation learning and clustering into one step. This approach uses a combined or joint loss function to merge the two stages into one. Such a loss typically consists of a representation learning loss and a clustering loss.

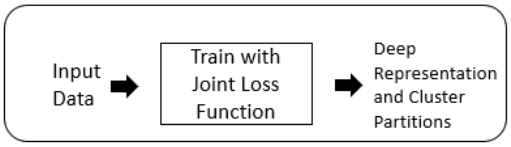


Figure 2.20: Joint deep clustering approach structure. Inspired by [71].

Closed-loop multi-step deep clustering is similar to multi-step sequential deep clustering. Like in multi-step, the approach consists of a representation learning and a clustering step. The difference is that the steps alternate in an iterative loop.

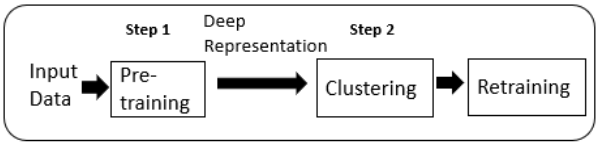


Figure 2.21: Closed-loop Multi-step deep clustering approach structure. Inspired by [71].

An important part of deep clustering algorithms of any category is the loss function. Different categories utilize different kinds of losses. Methods based on multi-step sequential deep clustering have a reconstruction loss and a clustering loss. The reconstruction loss is used to learn the deep learning method, and the clustering loss to learn the clustering method.

An example of a multi-step deep clustering method is VAMB[1]. The loss of VAMB is a composition loss of three terms. These terms are the abundance error E_{ab}, the composition error E_{TNF}, and a penalization D_{KL} term. The overall loss is defined as:

$$L = W_{ab}E_{ab} + W_{TNF}E_{TNF} + W_{D_{KL}}D_{KL} \tag{2.27}$$

The W denotes the weights of the terms.

The reconstruction error consists of the abundance error (E_{ab}) and composition error (E_{TNF}).

$$E_{ab} = \sum \ln(\mathbf{A_{out}} + 10^{-9})\mathbf{A_{in}} \tag{2.28}$$

$$E_{TNF} = \sum (\mathbf{T_{out}} - \mathbf{T_{in}})^2 \tag{2.29}$$

Here, the abundance error is defined as the cross-entropy and the composition error as the sum of squared errors. $\mathbf{A_{in}}$ and $\mathbf{T_{in}}$ are the abundance and composition inputs vector of VAMB. $\mathbf{A_{out}}$ and $\mathbf{T_{out}}$ are the embeddings of abundance and composition.

For regularisation purposes, the D_{KL} was taken into account as penalization term.

The term is defined as:

$$D_{KL}(latent|prior) = -\sum \frac{1}{2}(1 + \ln(\sigma) - \mu^2 - \sigma) \tag{2.30}$$

In this context, σ is the covariance and μ is the mean of a normal distribution N described by $X \sim N(\mu, \sigma^2)$.

Methods from the joint deep clustering category combine the objective of representation learning and clustering into a single one. Through that combination, it is possible to jointly learn and optimize the feature representations and clusters. The combined loss aims for suiting the reconstruction while considering the grouping.

An example of a joint clustering loss is the following loss[72]:

$$L = \lambda L_r + (1 - \lambda)L_c \tag{2.31}$$

Here, λ is a balancing hyperparameter, L_r is the loss of the representation learning and L_c is the clustering loss.

Another loss category is the cluster assignment hardening loss. A cluster assignment hardening loss is used in the case that the deep clustering method is based on cluster soft assignment. Methods that utilize such a loss have cluster assignment probability distribution and auxiliary target distribution.

2.7 Joint Deep Clustering

Joint deep clustering, also known as combined or end-to-end deep clustering method, jointly optimizes both representation learning and clustering [70]. The main advantage compared to the sequential deep clustering is the ability to iteratively learn such embeddings, that are optimal for the clustering task. When the embeddings are learned in a separate step as it is the case in sequential deep clustering, there is a risk that the embeddings are not the most suitable for the clustering task that is to follow. For that reason, by choosing an architecture that utilizes joint deep clustering, we hope to obtain higher-quality clusters.

The most popular network choice for joint deep clustering are stacked and convolutional autoencoders, described in Section 2.4. Nevertheless, some methods learn their embeddings with feedforward neural networks and deep belief network [70].

Joint deep clustering methods can be further classified into three main groups: the pretraining with fine tuning, the pretraining with joint training and the joint training [70].

Pretraining with finetuning means, that the network is first pretrained with the reconstruction loss only. Then in the finetuning part, the clustering loss is used to slightly manipulate the embedded space of the initialized model.

The pretraining with joint training method also pretrains the network using only the reconstruction loss. However after the pretraining, both the reconstruction and the clustering loss are used to manipulate the initialized embeddings.

Joint training methods then refers to methods that skip the pretraining part and optimize the network using both the reconstruction and the clustering loss.

2.8 Joint Deep Clustering Methods

This subsection proposes an overview of several state-of-the-art deep clustering methods.

2.8.1 Joint Unsupervised Learning

JULE (Joint Unsupervised Learning) is a CNN and agglomerative clustering-based method that jointly learns the feature representations and clusters [73]. It is a method for image clustering. Agglomerative clustering is a hierarchical clustering method that first creates large number of smaller clusters and then merges them until k-clusters are reached. The clusters are merged based on affinity. Therefore an affinity matrix based on the embeddings is created. An affinity matrix keeps track of the probability that two data points are related. Further to mention is that JULE is implemented as an (RNN) recurrent neural network. In RNNs, the neurons of a layer are connected and can take information from prior inputs into account. Therefore they are usually used for sequential data. For a set of n input data $I = \{I_1, ..., I_n\}$, the global objective of JULE is defined as the following:

$$\arg\min_{y,\theta} \mathcal{L}(y,\theta|I) \tag{2.32}$$

Here, $\mathcal{L}(\dot{})$ denotes the loss function, y denotes cluster IDs, and θ is a parameter for representation learning. JULE breaks up the optimization into two alternating steps. The cluster step is defined as:

$$\arg\min_{y} \mathcal{L}(y|I,\theta) \tag{2.33}$$

The representation learning step is defined as:

$$\arg\min_{\theta} \mathcal{L}(\theta|I, y) \tag{2.34}$$

It performs clustering in the forward pass and representation learning in the backward pass. The benchmark on image clustering conducted in the JULE paper showed competitive results. For the benchmark, the performance of JULE was compared to e.g. k-means and NCuts. On the downside, the usage of affinity leads to high computational and memory costs on larger datasets.

2.8.2 Deep Embedded Regularized Clustering

DEPICT (Deep Embedded Regularized Clustering) is a CAE-based deep clustering method that utilizes joint clustering and reconstruction loss. The clustering loss is based on the KL divergence technique (Section 2.5.2) [70]. DEPICT is structured into softmax layers that are attached to a multi-layer convolutional autoencoder. The structure of DEPICT is illustrated in Figure 2.22.

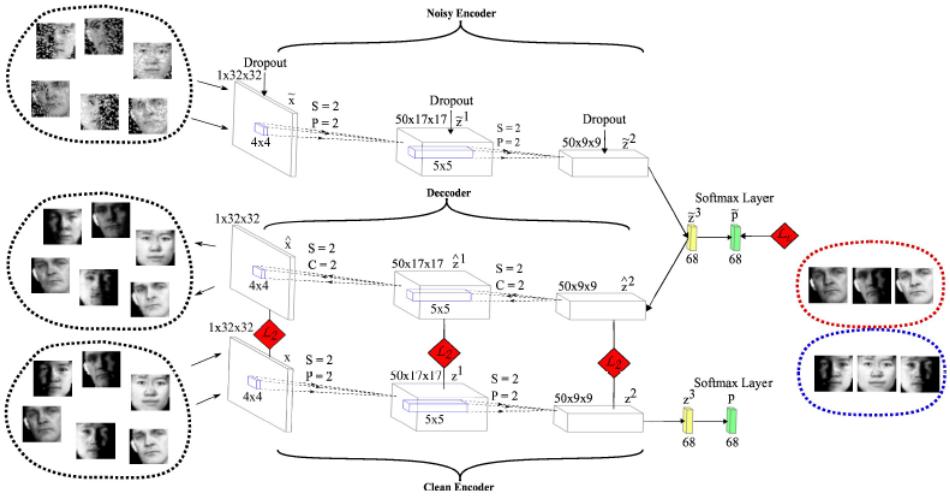


Figure 2.22: Visualization of the DEPICT structure [23].

DEPICT's convolutional autoencoder consists of an encoder layer, a decoder layer, and an encoder in the general structure. In the structure it is visible that a softmax layer is attached to each encoder. L_E is the clustering loss that is applied after the first softmax layer. The reconstruction loss L_2 is attached between the decoder and the second encoder. In overall it can be said that DEPICT jointly integrates the clusters by attaching the softmax layer in between the encoder and decoder part. DEPICT uses an entropy loss that contains a regularisation term for

the clustering. Regularisation is added to the clustering part to avoid outliners influencing the clusters. The benchmark on image clustering conducted in the DEPICT paper showed competitive results. For the benchmark, the performance of DEPICT was compared to e.g. k-means, NCuts, and JULE. DEPICT shows similar results to JULE. To note is that the structure of DEPICT is much more memory friendly than the one of JULE.

2.8.3 Deep Convolutional Embedded Clustering

Deep Convolutional Embedded Clustering (DCEC)[67] is a joint KL divergence clustering method that simultaneously uses both reconstruction and clustering loss for the task of unsupervised clustering. DCEC uses a convolutional autoencoder structure that learns embeddings of the input data. This is followed by joint cluster assignment and embedded feature refinement, until a stopping criterion is reached.

The method consists of two main phases. The first phase is parameter initialization, and the second phase involves parameter optimization and clustering. In the first phase, DCEC pretrains the convolutional autoencoder using only the reconstruction loss. This allows the network to a compute meaningful target distribution. In the second phase, finetuning is performed iteratively by using both KL divergence loss as a clustering loss L_c, and MSE as a reconstruction loss L_r. By incorporating the reconstruction loss, the local structure of the data is preserved and corruption of feature space is minimized.

The resulting joint loss minimization L is an optimization problem that is solved by a back propagation and stochastic gradient descent [67].

DCEC was benchmarked on image-based datasets MNIST and USPS, with very good results. Compared to JULE, DCEC performed about 2% worse on the MNIST dataset [73].

In Figure 2.23 is shown the structure of the convolutional autoencoder of the original DCEC model. Specifically, it shows the proposed structure for the MNIST dataset. Here, the autoencoder consists of three encoding layers Conv1, Conv2 and Conv3; and three decoding layers DeConv1, DeConv2 and DeConv3. In the middle there is an embedding layer h of size 10, followed by a fully connected layer FC [67].

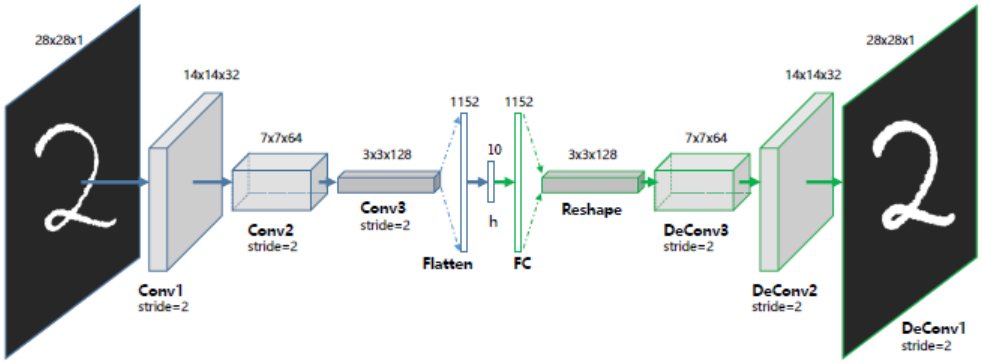


Figure 2.23: The structure of Convolutional Autoencoders in the original DCEC model for MNIST dataset [67].

2.8.4 Deep Embedded Clustering

Deep Embedded Clustering (DEC)[74] is one of the first methods that proposed joint clustering learning by simultaneously learning feature representation and updating cluster assignments using deep architecture. The proposed implementation of DEC uses a stacked autoencoder as the deep neuron network to learn feature representation as initialized parameters in the first phase. It is worth mentioning that the authors discard the decoder part after training and only uses the encoder with its feature representation. In the next phase, the clustering optimization is done by iteratively computing auxiliary target distribution and minimizing the KL divergence.

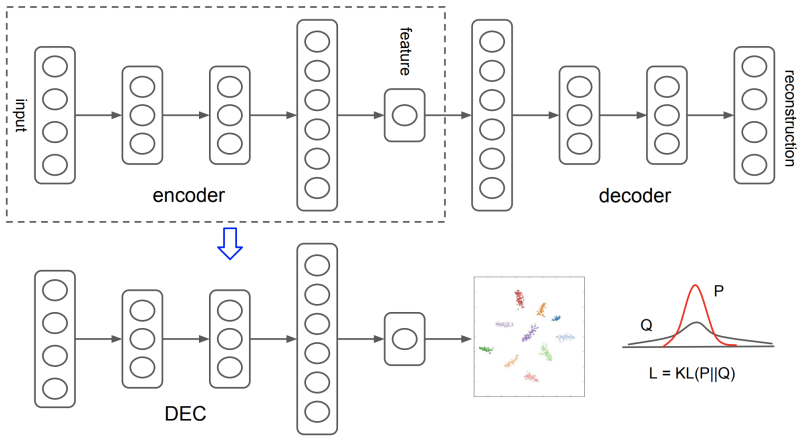


Figure 2.24: The overall structure of DEC [74].

The Figure 2.24 illustrates the DEC structure where we can see that the DEC

employs only the encoder part of the stacked autoencoder. It relies on the KL divergence loss of the Student's t-distribution P and target distribution Q to enhance clustering result. The detail of Student's t-distribution and target distribution calculation is clarified more in section 2.8.6. DEC is the pioneer [70] that applies joint learning in clustering, the method gives a lot of inspiration for next studies in the field of clustering using deep neuron networks.

2.8.5 Discussion: Method Selection

After evaluating different deep clustering methods, we have concluded that Deep Convolutional Embedded Clustering (DCEC)[67] is the best fit for our use case. In this project, the method will be adapted for the use in metagenomic binning. When selecting the most suitable method, we have considered several criteria. This includes computational resources required by the method, number of hyperparameters that needs to be tuned, the type of joint loss function used and the specific type of deep clustering that is utilized by the method. This section further elaborates on each of the criterion.

Firstly, it is important that the selected method is not too computationally heavy. Not only would such a method be impractical for the purpose of this project. Our goal is to develop a method that could be used for metagenomic binning in practise. Reasonable running time and consumed resources are therefore of a concern. Metagenomic binning is a relatively computational heavy task that processes a large amount of data. That is why these already existing problems could be further accelerated when using a computation heavy method. For this reason, we eliminated a candidate solution DEPICT [23]. This model requires long running time and requires usage of some computationally heavy libraries. Another potentially suitable method Convolutional Embedded Network (CEN) [75] is designed for processing extremely large amount of data. For instance, it is using Apache Spark, an open-source analytic engine for large scale data processing [76]. Based on our research, such design would not be efficient for the purpose of metagenomic binning. Computational expensiveness of heavier models would also cause another disadvantage. In case of eventually suboptimal results, it would be very hard to discover what caused them. It could be because of unsuitability of the CNN-based approach itself, or it could be because of the overall model composition.

Another criterion to consider is the amount of hyperparameter tuning required. The more parameters are there to be tuned, the more complicated and hard the tuning process gets. In practice, such a lengthy process would create a great constraint for the usage of the method for metagenomic binning. Therefore, we prefer a method with a smaller number of parameters. This criterion led to our elimination of a method JULE [73]. Even though this method meets many of our other

requirements, it is explicitly stated that tuning large number of hyperparameters is one of the method's disadvantages.

Next, we also consider how old the methods are. We have discovered that even the state-of-the-art methods in this field are a few years old, and despite their age their results are still considered as benchmarks. However, it is better if a method is newer. This is because of the continuous development of the end-to-end clustering methods and improved results overtime. An example of this can be seen when comparing the methods DEC [74] and DCEC [67]. DCEC is the successor of DEC and improves the original method by preserving the local structure of the embedded data. That leads to improved results because of less disturbed feature space. The end-to-end approach is improved in the newer DCEC by incorporating reconstruction loss into the joint function, and therefore we eliminate the less suitable DEC method.

The selected method also needs to support unsupervised clustering. When searching for methods that perform CNN-based end-to-end clustering, we have discovered that a lot of methods only support supervised learning or partially supervised methods [77]. Since metagenomic binning is an unsupervised clustering task, we had to eliminate those methods. Our requirement for the method was its utilization of end-to-end clustering, also known as joint clustering. However, joint clustering training can be further divided into two main categories. These are pretraining with joint training, and joint training with fine-tuning. Pretraining with joint training tends to outperform joint training with fine-tuning. Newer methods, including DCEC, often belong to the first category.

Lastly, we also consider the clustering approach used in the method. For example, we assume that agglomerative clustering might be less suitable. It might be hard to merge clusters together when the structure of sequences is complicated. Since updating the clusters is often tightly incorporated into the joint loss function of the method, changing the clustering method requires a major change. Ultimately, it is better to choose such method that utilizes the preferred type of clustering algorithm. For this project, we prefer KL divergence clustering. It is computationally more feasible compared to graph-based clustering. Further, it is more suitable for our data compared to k-means clustering. K-means works best on circularly distributed data points.

After evaluating all of our requirements, DCEC was selected as the arguably best choice out of the candidate models. One of our favourite features in this model is its utilization of KL divergence clustering. This method has a low complexity of $O(nk)$ where k is the number of centroids. That means that the method can scale well for larger datasets. Furthermore, DCEC consists of building blocks that can be modified and adjusted to the purpose of this project. Another advantage of DCEC is its manageable number of hyperparameters that are to be tuned. Additionally, the DCEC paper is highly cited, which is usually one of the indicators of a credible

paper. On the downside, DCEC requires a number of clusters to be defined as one of the model's parameters. Even though this problem presents an additional challenge to the project, there is a number of possible workarounds and solutions which are addressed in Section 3.1.2.

The following section will introduce the selected method DCEC in more detail.

2.8.6 DCEC Description

DCEC consists of a convolutional autoencoders (CAE) and a clustering layer. Even though the number of convolutional layers can be changed, the original DCEC model consists of three convolutional layers in the encoder part and three transposed convolutional layers in the decoder part. The clustering layer is attached to the last embedded layer of the CAE, containing compressed features of the original inputs.

Each data point x_i of a dataset X with $X = \{x_1, ..., x_n\}$ is mapped into an embedding z by a mapping function f_θ, so that $z_i = f_\theta(x_i)$. When the clustering process is initialized, each embedding z_i is mapped into a cluster soft label [67]. Soft labels are calculated using Student's t-distribution.

Then, a target distribution is calculated [67]. Using this information, the clustering loss L_c is computed as a Kullback-Leiber divergence between the soft labels and the target distribution. Hence, soft labels are being continuously recalculated.

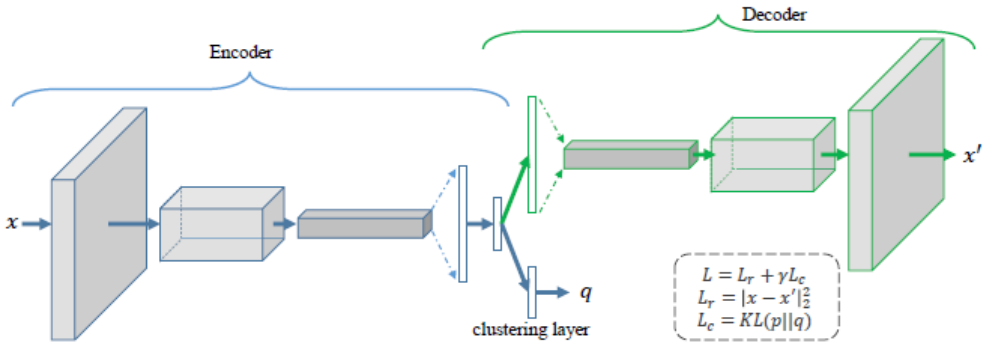


Figure 2.25: The overall structure of deep convolutional embedded clustering (DCEC) [67].

The overall aim of DCEC is to minimize the joint loss function L, defined as

$$L = L_r + \gamma L_c \tag{2.35}$$

where L_r is the reconstruction loss and L_c is the clustering loss. The coefficient γ is a value between 0 and 1 that sets the level of embedded space distortion that is allowed [67].

The reconstruction loss is based on the mean squared error loss (MSE) and it is defined as

$$L_r = \frac{1}{n}\sum_{i=1}^{n} \|g(f(x_i)) - x_i\|_2^2 \tag{2.36}$$

where the loss is calculated by taking the sum of squared differences between the inputs of the CAE encoder $f(x)$ and the outputs of CAE decoder $g(x)$. The resulting value is then divided by the number of input samples n.

Clustering using KL divergence

The clustering loss is defined as

$$L_c = KL(P\|Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}} \tag{2.37}$$

where the loss L_c is a Kullback-Leibler divergence between the soft assignment q_i and the target distribution p_i. KL divergence measures how far are two probability distribution from each other.

First, the method computes a soft assignment between the embedded features and initial cluster centroids assignments. Next, the mapping function and cluster centroids are updated [67].

Soft Cluster Assignment Mapping into soft labels is done by using Student's t-distribution. This technique was proposed as an effective solution for visualizing high-dimensional data into two-dimensional map [74]. Student's t-distribution is an infinite mixture of Gaussians, and it differs from Gaussian distribution by being more heavy-tailed. Evaluation of a point density is less computationally expensive when using Student's t-distribution compared to Gaussian, which is a major advantage of this approach [78].

The formula for the soft cluster assignment is defined as:

$$q_{ij} = \frac{1 + \|z_i - \mu_j\|^2)^{-1}}{\sum_j (1 + \|z_i - \mu_j\|^2)^{-1}} \tag{2.38}$$

where z_i is an embedded point i, μ_j is a centroid of cluster j and q_{ij} can be read as the probability that embedding i belongs to a cluster j [74].

Target Distribution The model is minimizing KL divergence by matching soft cluster assignments with a predefined target distribution p_i. This target distribution is modelled so that the clusters are learning from high confidence assignments and throughout the clustering iterations, the process should lead to stronger predictions. Additionally, the loss of individual centroids should be normalized to

prevent feature space distortion. This way, the model is able to train on its own high confidence predictions in an unsupervised manner [74].

The formula for target distribution is defined as:

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_j (q_{ij}^2 / \sum_i q_{ij})} \tag{2.39}$$

where p_{ij} is the target probability of a sample i belonging to cluster j. $f_j = \sum_i q_{ij}$ is the soft cluster frequency, computed as the sum of all soft assignments for that cluster.

Optimization of DCEC

Firstly, the model is pretrained with $\gamma = 0$. This is important in order to get a meaningful target distribution. After pretraining, the cluster centers are initialized by using k-means algorithm. Afterwards, γ is set to 0.1 and the process of updating CAE weights, centroids and target distribution starts [67].

The weights of the CAE and cluster centroids are updated jointly using Stochastic Gradient Descent (SGD). The formulas for computing the gradients for clustering loss L_c with respect to embedded point z_i and cluster centroid μ_j are defined as follows [74]:

$$\frac{\partial L_c}{\partial z_i} = \frac{\alpha + 1}{\alpha} \sum_j (1 + \frac{\|z_i - \mu_j\|^2}{\alpha})^{-1} \times (p_{ij} - q_{ij})(z_i - \mu_j) \tag{2.40}$$

$$\frac{\partial L_c}{\partial \mu_j} = \frac{\alpha + 1}{\alpha} \sum_i (1 + \frac{\|z_i - \mu_j\|^2}{\alpha})^{-1} \times (p_{ij} - q_{ij})(z_i - \mu_j) \tag{2.41}$$

where α is the learning rate.

After each few iterations, a target distribution is recalculated using the new soft cluster assignments. In every iteration, a parameter δ_{label} monitors the percentage of changed label assignments in between previous and current iteration. If the parameter is lower than a given threshold, then the training terminates. DCEC sets the threshold to 0.01 [67].

Chapter 3

Methods and Implementation

The previous chapter describes the principles of deep clustering and deep learning architectures. In this chapter, we describe how the architecture of the deep clustering framework DCEC can be modified to suit the task of metagenomic binning.

In this project, we have explored two versions of end-to-end metagenomic binners. Both versions are based on the architecture of the DCEC model (Section 2.8.3). The difference between the two is in the type of autoencoder that the models are using. Each of the models also takes a different type of input:

1. Deep Convolutional Metagenomic Binner (DCMB)

   This model is using convolutional autoencoder and takes as input raw sequential data - contigs. The model does not use any features obtained through preprocessing, such as abundance and composition. The goal is to discover, whether it is possible to learn features important for successful binning from unprocessed reads alone, with the use of convolution. We also explore to which degree joint clustering can improve our results.

2. Deep Stacked Metagenomic Binner (DSMB)

   This model is using stacked autoencoder, and takes as input widely used engineered metagenomic features - composition and abundance. Here, the main goal is to discover, whether joint centroid-based clustering (Section 2.6) will lead to more genomes being recovered, as opposed to two-stage clustering consisting of a pretraining step and a k-means algorithm.

The remaining part of this chapter describes both solutions in detail. First, we present the DCMB solution and finally DSMB. For all these solutions, we first describe the architecture of the model, together with selected implementation details. Next, we introduce experiments and finally their evaluation.

3.1 Deep Convolutional Metagenomic Binner (DCMB)

This section describes the general architecture and idea behind our proposed metagenomic binner Deep Convolutional Metagenomic Binner (DCMB). First, we will introduce the basic structure of the model. Then, we continue by explaining selected topics in more detail. Those are analyzed and assessed in Section 3.1.3.

3.1.1 Why a Convolutional Model?

There are two classic groups of deep learning methods applicable to sequential data: recurrent neural networks (RNN) and convolutional neural networks (CNN). Recurrent neural networks are predictive artificial neural networks with the capability to process and interpret temporal and sequential information. In RNNs, the neurons of a layer are connected and can take information from prior inputs to effect output, or even the current input. This create an illusion of a 'memory' [79].

Traditionally, sequential data is used with RNN [79], and CNNs are generally associated with image data [54]. However, the answer is not always that simple. The specific use case and overall goals determine which method is most suitable. Nowadays, a number of methods uses CNNs when working with sequential data. We can mention for instance the CNN-MGP method [13].

The reasons behind that can differ. For instance, RNN takes the absolute position of the detected patterns into account (i.e. chatbot generates a suitable answer based on the context of a whole conversation), while CNN primarily detects the local position of patterns (i.e. in face recognition, the model can recognize a human eye that can be found in any part of the image). In metagenomic binning, the reads or contigs are fragmented. Unlike other sequences such as sentences, there is no logical beginning and end in a single contig. In alignment-free methods, it is not known what should be the absolute position of the fragment and the detected patterns. For that reason, RNN methods seem less suitable for the problem domain of this project. Using information about the absolute position of the pattern will not be beneficial. On the other hand, metagenomic binner might hugely benefit from detecting patterns locally.

Secondly, CNN networks have been successfully applied for a gene prediction problems [13, 80], where hierarchical features have been detected by CNN-based neural network. Therefore we have an evidence, that CNNs have the ability to learn meaningful feature representations from DNA sequences.

Based on the arguments in this section, we have decided to use CNNs in the network architecture of our DCMB binner.

3.1.2 DCMB Architecture

An end-to-end clustering framework combines a network architecture with a joint clustering method. Both the network and clustering methods should be selected based on the intended usage of such a framework. In our case, we wanted to create embeddings from raw contigs.

Clustering techniques that we considered are k-means clustering, graph-based clustering, and KL divergence clustering. Choosing from those approaches, k-means clustering is not very suitable to apply for metagenomic data, because it tends to create round clusters. This might lead to poor results because metagenomic clusters are not characterized by round shapes. Graph-based clustering is a newer and interesting approach, not yet described in the scientific literature for metagenomic binning. The downside of graph-based clustering is that an enormous amount of computational resources is required for this method. As metagenomic datasets are often several 100 gigabytes big [4], this approach is not suitable. Lastly, there is the KL divergence clustering approach (Section 2.5.2). Even though this method initially instantiates the clusters using the k-means algorithm (Section 2.5.1), the method then continuously refines the clusters by using technique that shouldn't tend to round the clusters. We have decided to use this method because we assume that it is compatible with metagenomic data.

In conclusion, we want the DCMB binner to consist of a convolutional autoencoder and utilize the KL divergence joint clustering method. This combination of network and clustering methods is not new. In the recent years, several such methods were successfully applied for clustering image datasets. The best known examples include the DEC method [74], the DCEC method [67], and DEPICT [23]. Since the desired combination is already described in scientific literature, we decided to base our implementation on one of the existing solutions, namely the Deep Convolutional Embedded Clustering method (DCEC), described in Section 2.8.6.

To apply this method for a very specific case of metagenomic binning, numerous adaptations of the approach are needed to make it possible.

The Convolutional Autoencoder of DCMB

The structure of the convolutional autoencoder of DCMB is illustrated in Figure 3.1.

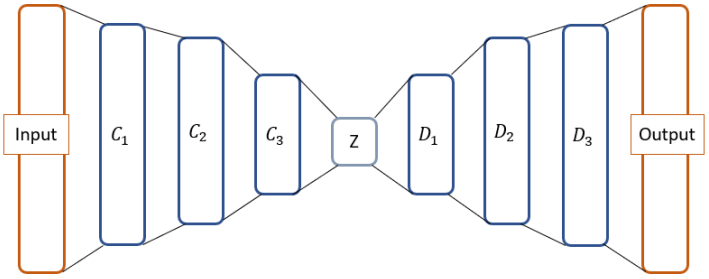


Figure 3.1: The overall structure of the convolutional autoencoder used for the DCMB.

The input of the autoencoder are one-hot encoded sequences that are padded with empty vectors so that they are all of the same lengths. This process is described in detail in Section 3.1.3.

The encoding part of the autoencoder consists of three sparsely connected 1D convolutional layers C_1, C_2 and C_3. The decoding part consists of three 1D convolutional transpose layers D_1, D_2 and D_3, also known as deconvolutional layers. In the middle, there is a fully connected layer z. The number of neurons in z is equal to the number of clusters n.

The goal of a deconvolution is to upsample the input into the output of a given dimensionality while using learnable parameters and preserving a connectivity pattern [81].

1D convolution differs from 2D convolution mainly in the direction of the convolution. This refers to the direction in which filter window size (kernel) moves. In 1D convolution, the kernel moves in a single horizontal direction from left to right. This is demonstrated in Figure 3.2 This makes this method suitable for time forecast data and sequential data [82].

Figure 3.2: The direction of 1D convolution.

In 2D convolution, the kernel slides in two directions, down the matrix. This is shown in Figure 3.3. It is especially well fitting for image datasets.

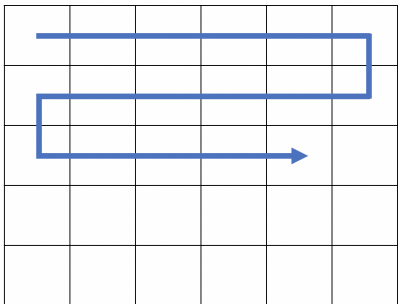


Figure 3.3: The direction of 2D convolution.

The gene prediction method CNN-MGP [13] successfully uses connected convolutional layers. This method one-hot encodes DNA sequences and utilizes 1D convolutional layers to obtain embeddings. This method inspired us to use convolutional layers for the metagenomic binning task.

However, there is a major difference between the goal of CNN-MGP and DCMB. CNN-MGP is solving a supervised learning problem. DCMB is solving an unsupervised learning problem. The network has to learn to create meaningful features without any external help. Consequently, this task is harder for the model to learn than the gene prediction task, given the complexity and difficulty to learn useful feature representation for metagenomic data.

More details about the network's parameter are described in Section 5.2.2.

The Joint Deep Clustering of DCMB

The overall architecture of the DCMB is visualized in Figure 3.4.

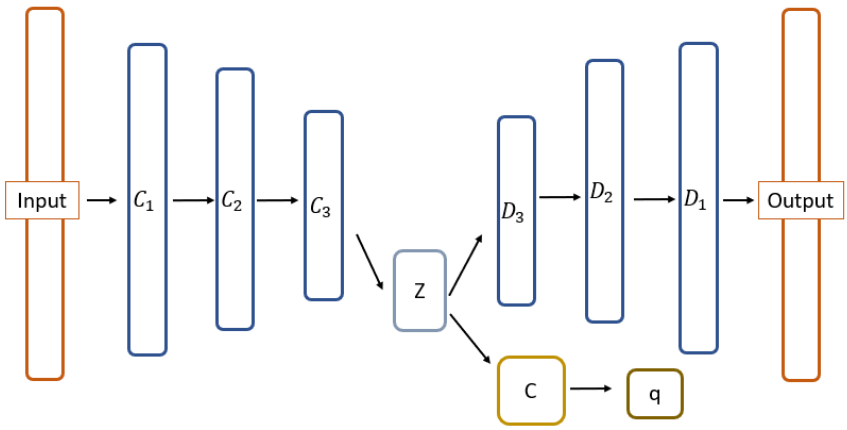


Figure 3.4: Architecture of the DCMB.

Figure 3.4 shows that the whole DCMB structure is composed of the previously described convolutional autoencoder and a clustering layer C attached to the embedding layer z of the autoencoder. This clustering layer maps each embedded point into a soft label q.

The clustering layer and the clustering loss L_c are borrowed from the DEC [74], respectively DCEC method [67]. The clustering loss is the KL divergence between soft labels Q and the predefined target distribution P:

$$L_c = KL(P\|Q) \tag{3.1}$$

Soft labels Q are the cluster assignments of embedded points in each iteration. The predefined target distribution P refers to the ideal clusters under the defined conditions, creating a soft constraint. These conditions include assigning a higher probability to the embedded points that are closer to the cluster centroids. Details of this clustering technique are described in Section 2.8.3.

The reconstruction loss of the convolutional autoencoder is calculating the cosine similarity between the original input vectors of y and the vectors $p(y)$ predicted by the decoding part of the autoencoder:

$$L_r = \frac{Y \times p(Y)}{|Y| \times |p(Y)|} \tag{3.2}$$

The final loss of DCMB is the sum of the reconstruction loss of the autoencoder, and the clustering loss:

$$L = L_r + \gamma L_c \tag{3.3}$$

γ controls the weight of the clustering loss in the final loss. DCMB sets $\gamma = 0.1$. If the value was higher, the clustering loss would effect and modify the embedding space too much, which could lead to worse clustering results [67].

The DCMB is first pretrained using only reconstruction loss. This lets the model create a meaningful initial target distribution. After that, the whole end-to-end model is trained using the compound loss L.

Downsampling Techniques One of the problems we had to solve is finding the best way to compress data throughout the convolution. There are two common approaches: max pooling and increasing the step size of the convolutional kernel, the so-called stride. While max pooling is a fixed operation, the stride is a parameter of a convolutional operation that can be learned. Since we want to maximize the learning capability of our model, we choose to use stride = 2 in all convolutional and deconvolutional layers. Setting stride > 2 could lead to lost information because the filter window size (kernel) would overlap fewer receptive fields.

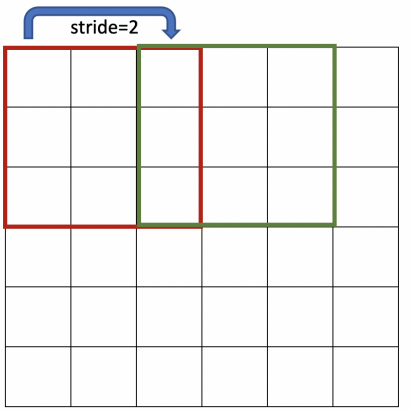


Figure 3.5: Step size (stride) of the convolutional kernel of size 3×3 is set to 2. In each step, the kernel moves 2 steps forward. The red square is representing the kernel in step x, green square is the kernel in step $x + 1$. There is less overlapping in between 2 consecutive steps, compared to stride=0.

Reconstruction Loss for Sparse, One Hot Encoded Data It is common for autoencoders, including convolutional autoencoders, to use mean squared error (MSE) as their reconstruction loss[67]. However, reconstructing one-hot encoded data is a challenging difficult task, and using a common loss function will often lead to poor results. The following problems are associated with using one-hot encoded data in autoencoders.

- MSE assumes that data are disjoint, but they are not. In each row, we want a single value to be 1 and the rest zeros. Therefore each row is a local multi-classification problem.
- The predominance of zeros can result in the decoder reconstructing all values as zeros because this will minimize the loss function. This can happen especially if we use unsuitable loss functions such as MSE or Cross-Entropy.
- In our experience, the decoder tends to reconstruct all values around 0.5.

The problem can be addressed by using such a loss function that will fit the purpose and address the issues listed above. For DCMB, we experimented with different loss functions, including a custom loss. The best results were achieved with cosine similarity loss (Section 2.2), which was our final choice. Literature also recommends Dice loss [83], which however didn't lead to good results in our model.

Ideally, we would treat the case as a multi-class classification problem, where each row representing one encoded nucleotide would be a multi-classification sub-problem solved with a cross-entropy loss. The results would then be combined.

Using cosine similarity loss, by comparing the original vector with predicted vector we could see that the largest predicted value in each row usually matched the original value 1:

Original vector	[0, 0, 0, 0, **1**]
Predicted vector	[0.15, 0.15, 0.25, **0.45**]

Table 3.1: Example of reconstruction of original input vector by the decoder of DCMB.

However, the optimal loss function should push the highest value closer to 1 and the rest of the values closer to 0.

Number of Bins Estimation

One of the challenges that we were facing is that in DCMB, the number of clusters k must be known beforehand. In metagenomic binner the number of bins is only known beforehand for synthetic datasets. Therefore it is required to estimate the number of bins from the input data before the binning.

One of the popular methods for finding a promising estimation of the number of clusters is the elbow method. This method is usually used in combination with k-means clustering. In the elbow method, the clustering algorithm is ran for a range of a possible numbers of clusters. For each of the runs, the degree of distortion is computed. The degree of distortion is the sum of squared errors between cluster centers and cluster points [84]. The main idea of the elbow method is that the first cluster will heavily decrease the distortion. In each run we will increase the number of clusters until we reach a point where the distortion will decrease steadily. The ideal number of clusters is the breaking point between the heavy and steady decrease of the distortion [84]. To find the ideal number of clusters, the distortion for all clusters within the range is plotted and the best number is chosen manually. An example for the elbow method is illustrated in Figure 3.6.

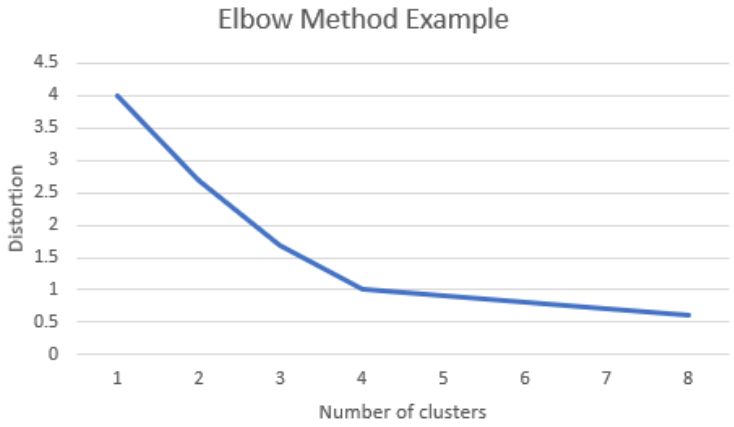


Figure 3.6: Example of the elbow method, where the ideal number of clusters determined by the method is 4.

The advantage of the elbow method is that the optimal number of clusters can be reliably estimated. The downside is that the method takes time, because the method has to run with all possible numbers of clusters within the given range.

For metagenomic binning, a popular way to estimate the number of clusters is to determine the number of single-copy marker genes [3] [85]. Marker genes are DNA sequences with a known position on the chromosome [29]. A single-copy marker gene is a marker gene that occurs once in a genome. To determine the number of single-copy marker genes, SolidBin[3] and COCOCOLA[85] use FragGeneScan and Hmmer. FragGeneScan is a sequence analysis tool used to find genes in short-read sequences [86], and Hmmer is a sequence analysis tool to search sequence databases that can be used for gene identification [87]. That means that FragGeneScan is ran with the FASTA file of the dataset as input. FragGeneScan tries to identify the single-copy genes in the dataset and append the found genes to a list. If FragGeneScan doesn't find an entry, which can be the case for long-read datasets, Hmmer analyses the input FASTA file and try to find database entries that match the given DNA sequences. Like in FragGeneScan, found genes are appended to a list of results. The estimated number of bins returned by this approach is the number of items in the list of results. The advantage of this method is that it can quickly estimate a promising number of clusters. The downside is that the estimation is not as precise like e.g. the elbow method.

We choose to use the single-copy marker genes-based method to estimate the number of clusters. The reason is that it can provide a good estimate without consuming too much time. DCMB utilizes the same implementation of the single-copy genes estimation as SolidBin and COCOCOLA.

3.1.3 Data Preprocessing

The general idea of the DCMB approach is an end-to-end metagenomic binner that operates directly on unprocessed contigs. In the context of this paper, by unprocessed contigs is meant their string representation. This is how they are stored in a FASTA file. An example of such a string is shown in Figure 3.7

```
GTGGAGTAGAAGAATTGAGAGCCTTATCAGTCAAAATAACAATTTAAGCTAT
```

Figure 3.7: String representation of a sequence fragment, as stored in FASTA file.

We need to feed these sequential data into a convolutional autoencoder. However, the sequences cannot be loaded into the model in their original form. Convolutional layers require numeric input, but our inputs are DNA strings. Another issue is the varying length of the sequences. Therefore, there are two main tasks to solve regarding the inputs. Firstly, it is needed to transform the sequences so that the autoencoder is able to learn meaningful representations. Secondly, it is required for the input sequences to be of the same length.

Input Transformation

There are several ways of how to transform sequential metagenomic data into format suitable for input into a convolutional layer. The most frequently used method is one-hot encoding of the sequences [13]. However, some other methods consider encoding DNA sequences into picture-like tensors [88], using graphical representation of DNA sequences [89] and ordinal encoding [90].

One-hot encoding One of the inspirations for this approach is CNN-MGP [13]. CNN-MGP is an approach for metagenomic gene prediction that predicts genes directly from raw DNA sequences. The approach is based on CNNs. Unlike metagenomic binning, the input data are ORFs (open reading frames) instead of contigs. ORFs are the area of a genome that contains amino acids [91], and each ORF is extracted from 700 base pair fragments. The string representations are one-hot encoded in a preprocessing step.

One-hot encodings are a way to represent categorial data in a numeric representation. A single one-hot encoding is an array of size n, where n is equal to the number of categories. In our scenario, each sequence is one-hot encoded into a matrix of dimensionality [length of the sequence, 4]. The 4 refers to the 4 types of DNA nucleobases: A, T, G and C. In this representation, each row in the matrix

contains three zeros and one value that is equal to 1 and represents the specific category. For our scenario of DNA nucleotides, the following table would represent a valid one-hot encoding scheme:

A	1	0	0	0
C	0	1	0	0
G	0	0	1	0
T	0	0	0	1

Table 3.2: Example of a valid one-hot encoding scheme for DNA-nucleotides.

Ordinal Encoding Ordinal encoding is another method to encode data. This method was successfully applied for encoding of DNA sequences by Choong et al. [90].

Instead of the binary array encodings of one-hot encodings, a single categoric data point is mapped to a specific numeric value. It is important to assure, that the numeric distances between all values are the same. For our scenario of DNA nucleotides, the following table would represent a valid ordinal encoding scheme:

A	0.25
C	0.5
G	0.75
T	1

Table 3.3: Example of a valid ordinal encoding scheme for DNA-nucleotides.

Other Encoding Methods There are also other, less common approaches to sequence encoding. For instance, Yin et al. [88] takes one-hot encoding of vectors a step further, and transforms those vectors into an image. First, the k-mers of length $k = 4$ are one-hot encoded. Since there exist 256 unique combinations of 4-mers, this gives them vectors of length 256. Each one-hot encoded vector of length 256 is representing one image pixel. Finally, the authors transform this data into square images [88].

Another graphical representation of DNA sequences was employed by Arias et al. [89]. The authors also transform the original DNA sequence into square images. Here, each k-mer in a sequence is represented by one pixel of a corresponding intensity [89].

We have found these graphical solutions very interesting. It is a unique way of transforming sequential data into a format that CNNs show best performance with - images. However, we have decided not to choose any of these methods, because there is not enough evidence whether this would work for our specific use case.

Conclusion One-hot encoding is not recommended for a large categorial space. The reason for that is that the encodings would have a huge size leading to the curse of dimensionality. Curse of dimensionality is a situation where with increasing size of data, the requirements for computational resources grow exponentially. This would become a problem, if we chose to one-hot encode k-mers of size >1. Since we decided to encode each nucleotide individually, we only have 4 distinctive classes.

An issue with ordinal encodings is that an order relationship between the categories is created. Through the numeric values it is assumed that in our example G < T, which is not the case. That ordering might lead to misleading values due to the mentioned relationship that is created.

Because of the categorial space for our data is only 4, and there is no relationship and order between the categories, we decided to use one-hot encoding method.

Handling Missing Values

In order to one-hot encode the data, additional preprocessing steps are required. In general the sequencing process can distort the data. If that happens and the nucleotide cannot be identified, the raw signal is translated into an error value *N* [92].

There are two basic approaches on how to handle these error values. One common method is to clean up these error values by replacing them with the most common nucleotide within each sequence. Second solution is to treat the error values *N* as a fifth nucleotide. This would mean, that one-hot encoding method would encode five symbols instead of four. However, this additional fifth vector would be extremely sparse. This solution also causes the size of input data to increase by one fifth, which is inefficient. It is also inaccurate to treat the symbol N as a new nucleotide, since we know it is either nucleotide A, T, G or C.

Ultimately, we choose to replace error values N with the most common nucleotide within the sequence. This way, the size of the encoding space is four instead of five.

Handling Variable Input Lengths

Secondly, a standard CNN architecture requires that its input data have the same shape and size. However, in most metagenomic datasets, the length of contigs varies. This can be solved by applying padding and masking techniques. The exact input shape will be specific for every metagenomic dataset.

Padding Padding means, that given the longest sequence *s* in the dataset, all remaining sequences shorter than the sequence *s* are filled with selected artificial

value. Typically, this value is set to 0. Padding is used in situations, where all sequences need to have the same length. This is an input data requirement by some neural networks such as CNNs.

Masking The padded values are artificial and should not affect the loss and weight values of the model. This would result in worsened performance of the model. This is why a step called masking is applied. Masking is a method, that indicates which elements of a matrix or vector should not be used [93]. By doing that, the model is told to skip processing the data hidden by masking so that they don't affect the learning of the model.

Before applying padding step, it is important to check the dataset for outliers and the most common length range. If there are only few sequences of much larger size than the rest, padding all sequences to their length will cause performance issues. Based on this knowledge, the length range for which padding will be applied can be selected.

Solution Since we chose to use one-hot encoding, there are two ways how to pad the sequences. Firstly, it is possible to pad the sequences with an arbitrary value such as *O*, and then encode this value into its own column:

A	1	0	0	0	0
C	0	1	0	0	0
G	0	0	1	0	0
T	0	0	0	1	0
O (padding)	0	0	0	0	1

Table 3.4: Example of a one-hot encoding where padding has its own column.

Since padding doesn't have a unique value, it is not possible to apply a standard masking technique. Also, this approach also does not represent the data accurately, because it treats padding as a nucleotide.

This is why we decided to apply the following padding method:

A	1	0	0	0
C	0	1	0	0
G	0	0	1	0
T	0	0	0	1
(padding vector)	0	0	0	0

Table 3.5: Example of a one-hot encoding where padding is attached to the end as an empty vector.

Here, we append empty vectors [0.0.0.0.] to fill the difference between the length of a sequence and the maximum length. For fields containing the value 0,

the output of convolutional operation will also be 0. This means that the weights will not get affected.

3.2 Deep Stacked Metagenomic Binner (DSMB)

This section describes the general idea and architecture behind our proposed metagenomic binner Deep Stacked Metagenomic Binner (DSMB).

3.2.1 DSMB Architecture

The DSMB (Deep Stacked Metagenomic Binner) is our second metagenomic binner that we propose. Compared to the DCMB, there are two major changes. One is that we replace the convolutional autoencoder with a stacked autoencoder (Section 2.4.1), and the second is that abundance and composition (Section 2.1.4) are taken as input. Meanwhile, the joint deep clustering part (Section 2.6) remains the same. The joint clustering is done by using the KL divergence clustering method, described in Section 2.5.2. Instead of taking raw sequences as input, we decided to take abundance and composition as inputs, similarly to VAMB [1].

For each contig, the input vector consists of an abundance vector A of size s and a composition vector T of size 103. The vectors are computed from the BAM files (abundance) and the FASTA file (composition) of the dataset. Those two vectors are concatenated. The size of the abundance vector depends on the number of BAM files linked to a dataset, and this is in most cases less than 10.

The structure of the stacked autoencoder used for DSMB (Deep Stacked Metagenomic Binner) is visualized in Figure 3.8.

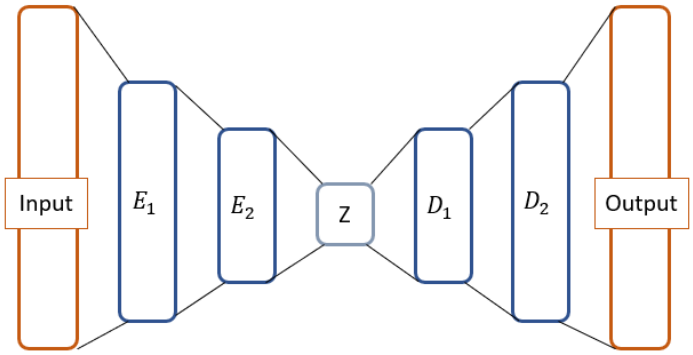


Figure 3.8: The overall structure of the stacked autoencoder used for the DSMB.

The stacked autoencoder consists of two stacked encoders and decoders, which were implemented through the use of Dense Layers, respectively fully connected

layers. Due to the relatively small size of the input vector, the size of the first encoder layer (E_1) and first decoder layer (D_1) is 64 neurons. The hidden space for the second encoder layer (E_2) and the second decoder layer (D_2) is of size 32. The size of the embedding (z) is 16. Each layer is batch normalized.

The end-to-end solution is similar to the DCMB (Section 3.1.2). The differences are that the DSMB utilizes a stacked autoencoder instead of a convolutional autoencoder and that it uses pre-processed composition and abundance vectors as input. The architecture of the DSMB is visualized in Figure 3.9.

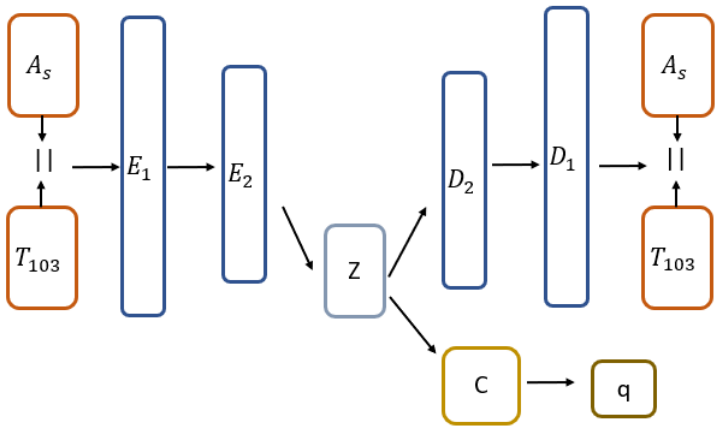


Figure 3.9: The overall architecture of the DSMB.

C represents the clustering layer. Z is the latent space of size 16, as mentioned. Each embedding is mapped into a soft label q. In contrast to VAMB, the output of the stacked decoder is the concatenation of abundance and composition vector and not the single abundance and composition vector.

Chapter 4

Datasets and Assessment Tools

4.1 Datasets

For our experiments, we have chosen two different datasets. One synthetic dataset with a ground truth, and one real-world dataset. *As we want to compare the results of our proposed binners with the results of our previous work [15], we use the same datasets that were used in our previous work. Namely, they are the CAMI Low dataset [4], and the Azolla dataset [5]. The following two subsections are rewritten versions from our previous work [15].*

4.1.1 CAMI Low Dataset

CAMI (Critical Assessment of Metagenome Interpretation) is an initiative to set a standard for evaluating metagenome analyses. The organization behind CAMI is the MICROBIOME Community of Special Interest, which aims for the advancement and evaluation of computer science methods in the area of microbiome research [94]. To evaluate the performance of a metagenomic binner, CAMI provides several synthetic datasets that are used by most state-of-the-binners for benchmarking [1, 2].

One of the lower complexity synthetic datasets that we already used in our previous work [15] is the CAMI Low dataset. The CAMI Low dataset is a synthetic short-read dataset that is part of the 1st CAMI challenge. It is a rather small dataset having a total size of 15 GB [4]. The data contains samples from 40 genomes and 20 circular elements obtained from the short read Illumina HiSeq [95]. Circular elements are e.g. viruses, and plasimids[95].

For evaluation, a synthetic dataset with a ground truth is an advantage, because it is possible to assess if the contigs or base pairs are assigned into a correct bin. We decided to use the CAMI Low dataset because it contains a ground truth, and the performance of the binner on the dataset can be conveniently evaluated by using

CAMI's assessment tool AMBER.

4.1.2 Azolla Dataset

As a representative of a lower complexity non-synthetic dataset, we chose the Azolla dataset. This dataset was created by Laura W. Dijkhuizen from the Utrect University as a subset of a larger dataset [5]. The dataset contains samples of the aquatic fern species Azolla filiculoides [7], and it contains DNA of the Azolla plant and several bacteria species that use the plant as habitat. A list with all organism that are contained in the habitat is visualized in Figure 4.1 [7].

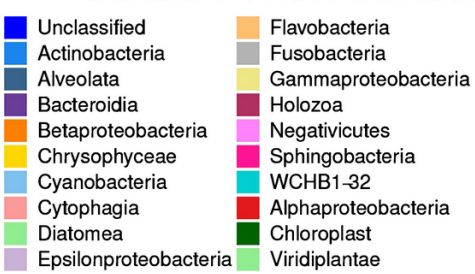


Figure 4.1: Organisms that are found in the habitat of Azolla.

Similarly to the CAMI Low dataset, this dataset is rather small with an overall size of 10 GB. Because this dataset is non-synthetic, there is no ground truth for evaluation. Nevertheless, the dataset can be evaluated by using the assessment tool CheckM.

4.2 Assessment Tools and their Measures

This section explores the assessment tools AMBER and CheckM and the measures used by these tools.

4.2.1 AMBER Overview

AMBER (Assessment of Metagenome BinnERs) is an assessment tool provided by the MICROBIOME Community of Special Interest. It was created as an evaluation tool for CAMI datasets. AMBER creates measures that illustrate the performance of a binner on a dataset, such as completeness and contamination. Because every CAMI dataset is synthetic and hence contains a ground truth, it is possible to assess which contig was matched to the correct bin by a given binner.

4.2.2 AMBER Evaluation Measures

Some of the most commonly used evaluation measures in metagenomic include completeness and contamination. Predicted bins, in this case the results of DCMB and DSMB clustering, are compared against the ground truth bins. For synthetic datasets it is known, which contigs belongs to which bin. These expected bins, also known as the gold standard, are the ground truth. In cases where gold standard is not known, other evaluation techniques have to be used instead.

AMBER tool maps each predicted genome bin into a single genome by two different means: base pairs (bps), and sequences (contigs).

- Base pairs

  Each bin is mapped to the genome that best represents the bin. This means, that majority of base pairs in the bin belong to that genome [96].

- Sequences

  Each bin is mapped to the genome that best represents the genome. This means that most of the genome is contained within the predicted bin [96].

Completeness

The completeness, also known as sensitivity, measures the percentage of contigs or base pairs that were expected to be found in a predicted bin and that were indeed found there. It is defined as the following:

$$r = \frac{TP}{TP + FN} \tag{4.1}$$

Here, TP (true positives) refers to all contigs/ base pairs that are correctly classified. FN (false negatives) are all contigs/ base pairs that belong to the bin but weren't assigned into it.

Contamination

The contamination evaluates the percentage of contigs/ base pairs that have been falsely classified into a particular bin. It is defined as the following:

$$r = 1 - \frac{TP}{TP + FP} \tag{4.2}$$

Here, FP (false positives) refers to all the bins/ contigs that were assigned into a bin, but don't belong to it.

Accuracy

The accuracy evaluates, to which degree were the base pairs assigned to the correct bin. This is calculated over the entire dataset. It is defined as the following:

$$a = \frac{\sum_{x \in X} TPx}{U + \sum_{x \in X} TPx + FPx} \tag{4.3}$$

Here, U is the number of unassigned base pairs, and x is a bin of the set of all bins X.

Another important measure of AMBER is the number of recovered bins. Recovered bins are all bins that recover a genome, meaning that the bin is less than 10% contaminated and more than 50% complete.

Rand index adjusted for chance (ARI)

Rand index adjusted for chance is a commonly used measure that compares predicted clusters against the correct clusters, in this case the gold standard. The general formula is defined as:

$$ARI = (RI - Expected\_RI)/(max(RI) - Expected\_RI) \tag{4.4}$$

Here, RI refers to the actual similarity of pair-wise comparisons between the predicted and expected results. $Expected\_RI$ is the expected similarity of these pair-wise comparisons. The maximum Rand index $max(RI)$ is typically set to 1. The score ranges between -1.0 and 1.0, where value close to 0 signifies random cluster assignment while a value close to 1 stands for a great similarity [97], [98].

4.2.3 CheckM Overview

CheckM is an assessment tool to estimate genome completeness and contamination [99]. Unlike AMBER, CheckM evaluates metagenomes not based on a ground truth but marker genes. Marker genes are DNA sequences where the position of the chromosome is known [29]. Therefore, CheckM is used for non-synthetic datasets. CheckM searches for marker genes that belong to a specific genome lineage. In this context lineage refers to the classification of an organism in the taxonomic rank. The taxonomic rank is visualised in Figure 4.2.

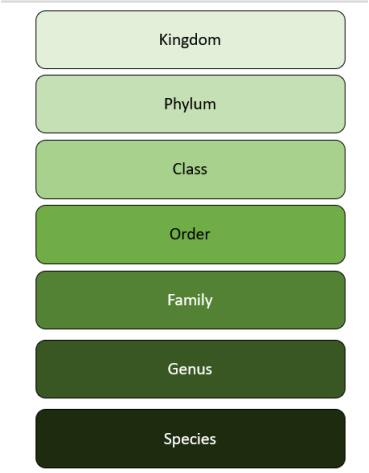


Figure 4.2: The taxonomic rank starting with kingdom. Inspired by [100].

The taxonomic rank goes from kingdom to species. An example for a kingdom is animals and an example for a species is the Homo sapiens. As some marker genes occur several times within a lineage, it is hard to tell which taxonomy the DNA sequence exactly belongs to. Therefore, CheckM makes use of marker sets. Marker sets are sets of marker genes. They are used as a certain combination of marker genes that occur only in a specific taxonomy.

4.2.4 CheckM Evaluation Measures

Like AMBER, CheckM computes the measures completeness and contamination. Because CheckM does not evaluate a ground truth, the measures are calculated differently from how AMBER does it.

Completeness

CheckM computes completeness by evaluating whether the marker genes that should be in the bin are its members. This is defined by the following formula:

$$\frac{\sum_{s \in M} \frac{|s \bigcap G_M|}{|s|}}{|M|}$$

M is the set of all marker gene sets that occur in the bin, and s is a single marker gene set of the set M. G_M is the set of marker genes that belong to the bin.

Contamination

CheckM computes contamination by evaluating the number of multicopy genes occurring for every marker gene set [99].

$$\frac{\sum_{s \in M} \frac{\sum_{g \in s} C_g}{|s|}}{|M|}$$

C_g is the value for the frequency of occurrence of a gene g. C_g is $N - 1$ with N being the number of occurrence if $N \geq 1$. If the gene is missing, the value is set to 0.

Chapter 5

Experiments and Evaluation

In this chapter, we explore the impact of joint deep clustering on metagenomic binning. In Section 5.2, we utilize DCMB to conduct several experiments on the CAMI Low dataset. With these experiments, we want to investigate how does joint metagenomic binning based on unprocessed contigs behave. We also want to explore the potential of using raw contig as the only input to our model.

Because the architecture and the results of the DCMB are different from the state-of-the-art binners, in Section 5.3 we utilize additional experiments using the DSMB model. The DSMB uses composition and abundance vectors as input. On the DSMB, we run experiments that give more insights into the impact of joint deep clustering on metagenomic binning. The DSMB experiments are performed on the CAMI Low and Azolla datasets.
The experiments frequently mention the following terms:

Epoch Training the model for one cycle with all data during so-called pretraining. Pretraining is an initial stage of the model training, where we only train the autoencoder part of the model. Clustering layer is not used during this stage. The goal of the pretraining is to get an initial meaningful feature representations. Only the reconstruction loss is calculated.

Iteration Training the entire end-to-end model for one cycle with all data after pretraining. In this stage, both autoencoder and clustering layer are used. Feature representations and clusters are learned simultaneously. Both the reconstruction and the clustering loss are calculated.

5.1 Experimental Setup

For all experiments, we are using a desktop PC provided by the AAU's IT department with the following specifications [15]:

Item	Value
OS	Ubuntu 21.10
Processor	11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz
RAM	32 GB
Graphics Card	GeForce RTX 3070 Ti
Hard Drive	1000 GB SSD

Table 5.1: Hardware and OS specification of the computer used for the experiments.

By not using a cloud server like the AAU intern CLAAUDIA or Google Cloud, we have to deal with constraints on speed and memory resources. Therefore our setup can handle smaller datasets better as bigger ones. From the previous project [15], we know that smaller datasets are sufficient enough to explore the general performance of a binner. Hence these limitations are not critical.

5.2 Deep Convolutional Metagenomic Binner (DCMB)

This section describes experiments that were done to evaluate the performance of the Deep Convolutional Metagenomic Binner (DCMB). The DCMB utilizes unprocessed contigs for the joint metagenomic learning. The architecture and further implementation details of this model are described in Section 3.1.2.

5.2.1 Experimental Scenarios with CAMI Low Dataset

The main focus of the experiments is to investigate the performance of a metagenomic binner that takes unprocessed DNA strings as the input and it is based on joint end-to-end learning. With the experiments, we want to find the optimal parameters and structure that would lead to the best results. Best results mean in this context, to obtain the highest purity, completeness, and accuracy. Ultimately, the most important indicator of the performance of a binner is the number of recovered genomes.

The experiments are designed with the goal to **explore whether convolution-based model taking only sequences (contigs) as input is a promising solution** for the task of metagenomic binning. We want to explore a convolutional metagenomic binner, as such a model has not been published in scientific publications yet.

Another goal is to **test the effectiveness of the selected joint clustering method, based on KL divergence clustering technique**. This approach refines clusters by slightly modifying the embedding space after initial pretraining by using both clustering and reconstruction loss Section 2.5.2.

The joint clustering with KL divergence is a state-of-the-art method that was tested and proven successful on a number of image datasets. In [67], the method is tested on MNIST and USPS datasets. On these datasets, this method utilizing joint

deep clustering achieved better accuracy than method that utilized plain k-means clustering [67], [23].

However, the performance of such a method is not documented for metagenomic datasets. Compared to the commonly used datasets such as MNIST, metagenomic datasets are much larger. They are also more complex, because of the number of motives that needs to be detected for a successful clustering.

Based on our goals, we have defined the following experimental scenarios:

1. **Base case and its benchmarks against existing binners.**

   For this scenario, we introduce our best result and compare it with results obtained using state-of-the-art binners.

2. **Analysis of the impacts and benefits of joint deep clustering.**

   This experimental scenario is to confirm if and how we can benefit from the joint deep clustering with KL divergence. Additionally, we show how the weight of the clustering loss affects the results of the joint deep clustering.

3. **Impact of the length of sequences on binning result.**

   This experiment explores the optimal length of one-hot encoded sequences, leading to the best results. Is it better to cut the sequences at a fixed length, or is it better to define a maximum sequence length and use padding to resolve the length differences? Will filtering out of very short contigs lead to better clusterings?

4. **Impact of the number of clusters on binning result.**

   How will the results differ, if we set the number of clusters n as the true number of genomes within the dataset, versus if we detect the number of clusters using the single-copy genes method from Section 3.1.2?

Each of the scenarios is further described and evaluated in the remaining part of this chapter.

5.2.2 Base Case

DCMB Base Case Setup and Hyperparameter Configuration The model setup is identical for all experiments if it is not stated otherwise in the experiment description. The decision for the setup and configuration is based on thorough testing. For the best result obtained (DCMB Best), this setup is as follows.

Both encoder and decoder consist of three convolutional layers. The kernel of those layers convolves over a single spatial dimension, which is well-fitting for our sequential data The lengths of the convolutional windows are 5, 3, and 3. The stride lengths of the convolution are set to 2. This is identical across all layers. The

numbers of output filters in the convolution are set to 32, 64, and 128. The number of clusters is set to the number of bins contained in CAMI Low dataset, which is 60. The embedding size is by default set to the number of clusters which is therefore also 60. The reconstruction loss is defined by the cosine similarity loss, explained in Section 2.2.1. The batch size is set to 256, and we use the Adam optimizer.

After each layer except the last one, we apply activation function ReLU. After the last layer in the decoder, we apply sigmoid activation function which scales the outputs in between the values 0 and 1.

Regarding the joint deep clustering setup, we train the model with 200 epochs, and 700 iterations. The target distribution gets updated after every 140 iterations. The tolerance threshold δ is set to 0.01. This means that if less than 1% of embedded points gets reassigned to a different cluster in between 2 consequent iterations, the training will be stopped. The weight γ of clustering loss L_C is set to 0.1.

Input preprocessing differs for every dataset. In order to decide on the optimal preprocessing steps for CAMI Low, we have done a data analysis of the dataset.

CAMI Low data analysis By analyzing the CAMI Low dataset, we found that 95% of the contigs contain less than 20000 nucleotides. Out of that number, 50% of contigs contain less than 1000 nucleotides, while 5 contigs include more than 1000000 nucleotides. That means that the length of contigs varies from very small to large. The distribution of the data lengths is visualized in Figure 5.1.

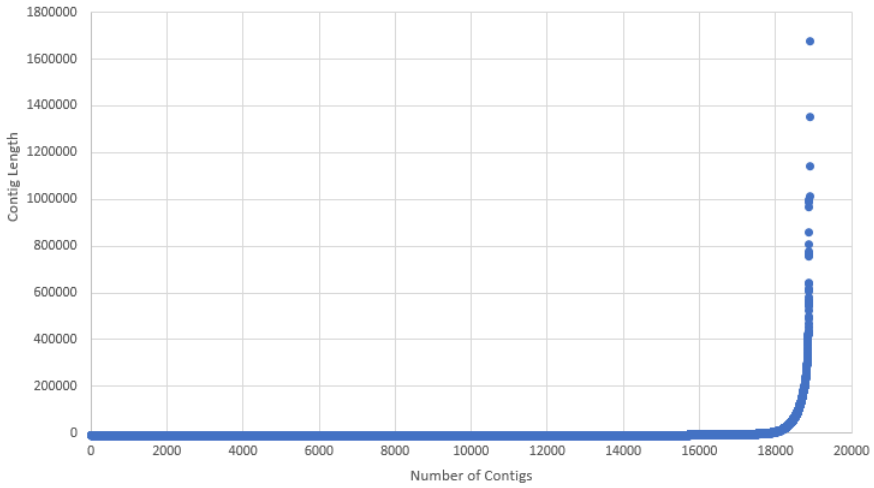


Figure 5.1: Distribution of the length of sequences in CAMI Low dataset.

We decided to select a common length range that will include most of the contigs. Since 95% of contigs are shorter than 20000, we consider the remaining 5% as outliers. We then cut those outliers at 20000 and use their first 20000 nucleotides. During input preprocessing, we also exclude all sequences containing less than 750

nucleotides. We have discovered, that it is hard for the binner to correctly cluster short contigs, and they add noise to the model.

Consequently, the length of the sequences ranges between 750 and 20000. Next, we transform those sequences by using the one-hot encoding technique described in Section 3.1.3. Finally, we are padding the one-hot encoded sequences using the technique in Section 3.1.3, resulting in the final input dimension (number of sequences, length of sequences, number of nucleotides to encode):

$$dim_{CAMILow} = (19679, 20000, 4) \tag{5.1}$$

5.2.3 Base Case and Its Benchmarks Against Existing Binners.

From the first look at the results it is clear that DCMB performs less well than the state-of-the-art binners, namely VAMB, SolidBin and MetaBAT 2.

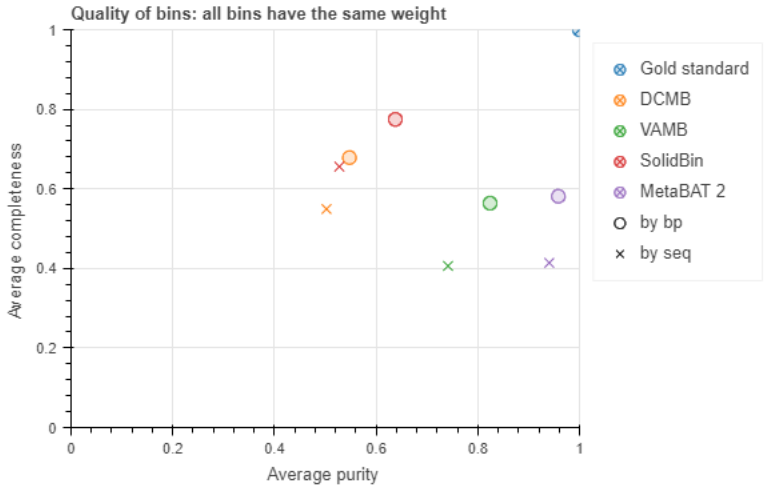


Figure 5.2: Plot representing the quality of bins on a bin level.

Figure 5.2 shows, that compared to the other binners, DCMB has the lowest average purity. Looking at the average completeness of 0.679, DCMB outperforms VAMB, SolidBin and MetaBAT 2 for both sequences and base pairs. Nevertheless, it is to mention that DCMB does not recover any bin.

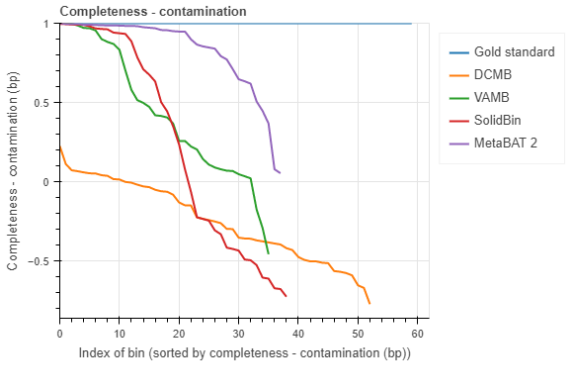


Figure 5.3: Plot representing the quality of bins based on a completeness - contamination metric.

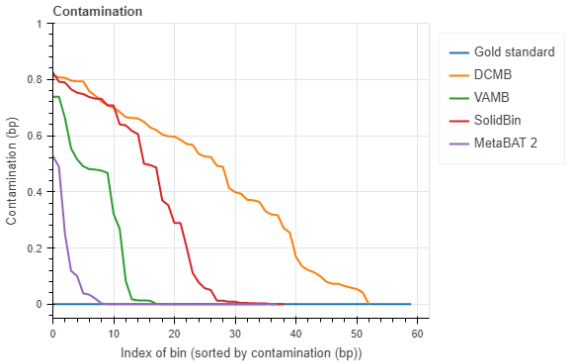


Figure 5.4: Plot representing the quality of bins based on a contamination metric.

Figure 5.3 illustrates the quality of bins, where the bins are sorted in a descending order from those with the highest completeness and the lowest contamination, to bins with the opposite qualities. None of the compared binners can compete with the qualities of the bins from gold standard. However, DCMB clearly underperforms in this evaluation, because the DCMB bins that have higher completeness are also more contaminated. On the other hand, bins with high low contamination have low completeness. This results in no genomes being recovered using DCMB Best. In order for a genome to be considered recovered, the corresponding bin has to be at least 50% complete and include less than 10% contamination.

Figure 5.4 shows the level of contamination per bin in a descending order. Even though DCMB creates bins with low contamination, the average level of contamination per bin is higher compared to the other binners. The descent of the DCMB curve is slower.

5.2.4 Analysing the Impacts and Benefits of Joint Deep Clustering

For this experiment we compare our best result from the joint training (DCMB Best) to the result from a two-stage version of the DCMB method (DCMB Pretraining + k-means). The two-stage method consists of a full pretraining stage, directly followed by clustering using a plain k-means clustering algorithm. That means that for the two-stage version, the deep clustering method is not used. The comparison also includes a variant of DCMB model, where after pretraining, we use only clustering loss for the end-to-end part of the training (DCMB Clustering Loss). That means, that after pretraining of the autoencoder, we disregard the decoder. According to [67], by letting the clustering loss to have so much effect on modification of the embedding space, this will result in major changes in the embedding space, which will no longer match the original data distribution. Consequently, worse results are

to be expected. We want to explore if this will be apply also for the metagenomic data.

The weight of the clustering loss in DCMB Clustering Loss increases from 0.1 to 1, and the weight of the reconstruction loss lowers from 0.9 to 0. This is inspired by the DEC method [74]:

$$L = 0 \times L_r + 1 \times L_c = L_c \tag{5.2}$$

To be comparable, all three results are obtained from the same test run. Therefore, the parameters of the model, as well as architectural details, are identical for all variants.

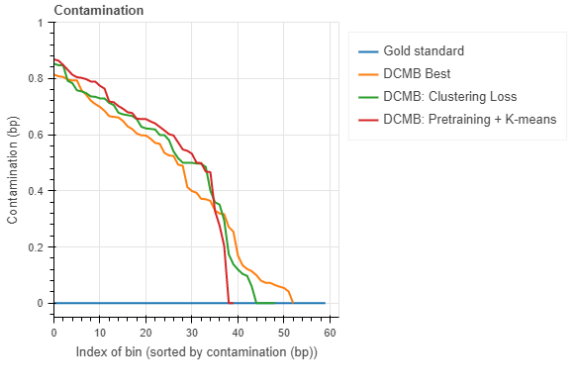


Figure 5.5: Plot representing the quality of bins based on a contamination metric.

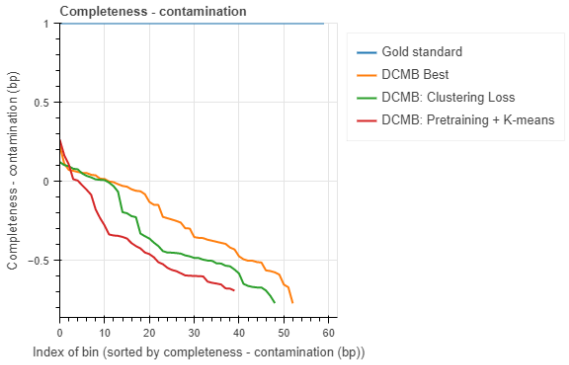


Figure 5.6: Plot representing the quality of bins based on a completeness-contamination metric.

The graph in Figure 5.5 shows the quality of bins considering their contamination, while Figure 5.6 shows both completeness and contamination of the bins. Even though the differences are not marginal, we observe visibly varied results. The first difference is in the number of clusters that were created. Two-stage DCMB (DCMB Pretraining + k-means) only created 38 clusters. The curve in both plots also indicates the lower quality of bins compared to the other two models. DCMB Clustering Loss, that trained with only a clustering loss is performing better than the two-stage DCMB, but worse than DCMB Best.

In conclusion, we have achieved worse results with DCMB Clustering Loss than we did with DCMB Best. This indicates, that by modifying the embedding space by using only clustering loss, the embedding space is altered too much, and it represents the original data less well. On the other hand, if we preserve the decoder part by keeping the reconstruction loss of the autoencoder throughout the whole training as it is done in DCMB Best, we create embeddings that are more optimal for the clustering task.

However, the experiment showed that DCMB Clustering Loss still outperforms the two-stage DCMB (DCMB Pretraining + K-means). That means that the learning benefits are stronger than the disadvantage of a slight corruption of the embedding space.

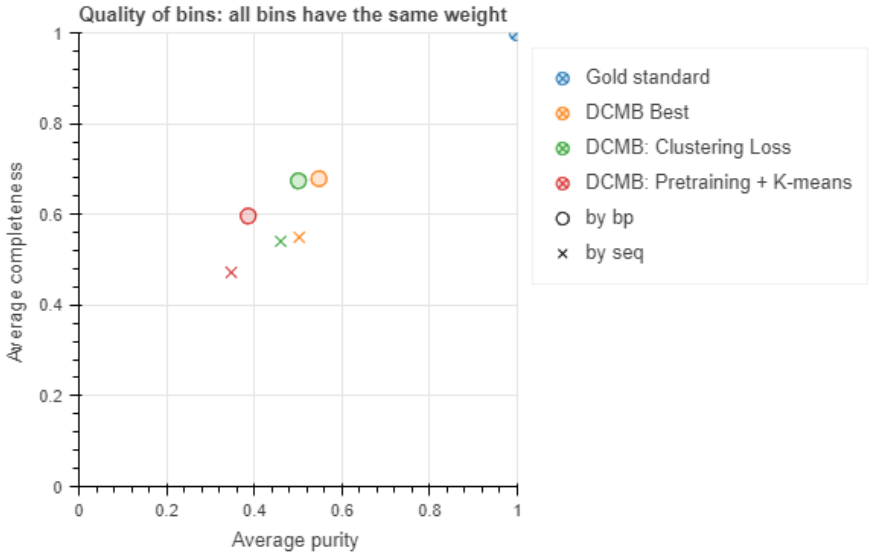


Figure 5.7: Plot representing the average quality of bins based on their average completeness and average purity.

The quality of bins in Figure 5.7 is quite comparable among the three models, with DCMB two-stage variant slightly falling behind the other two.

Binner	Accuracy (bp)	Av. purity (bp)	Av. completeness (bp)	ARI (bp)
Gold Standard	1	1	1	1
DCMB Best	0.33363	0.5475	0.67861	0.22901
DCMB Pretraining + Clustering Loss	0.25647	0.50073	0.67384	0.1473
DCMB Pretraining + K-means	0.28174	0.38616	0.59642	0.2008

Table 5.2: Comparison of accuracy (bp), average purity (bp), average completeness (bp), and adjusted Rand index (bp) between DCMB Best result, DCMB Pretraining + K-means and DCMB using only clustering loss in end-to-end training (DCMB Clustering Loss).

Finally, the overview of main metrics is shown in Table 5.2. Looking at these results, the better results of the two-stage clustering over DCMB Pretraining + K-means are less obvious. It is a bit surprising that the two-stage DCMB has a higher base-pair accuracy and a higher adjusted Rand index. However, the end-to-end method using only clustering loss has significantly higher average purity and completeness. The inconsistency in the results might be caused by the DCMB

two-stage having only a few bins with high completeness and purity, while DCMB with clustering loss has bins of higher quality on average.

In conclusion, DCMB Best achieves the best results out of the three compared models. Therefore, we can see a positive impact of the joint deep clustering with KL divergence method, resulting in the highest accuracy, purity and completeness metrics.

5.2.5 Impact of the Length of Sequences on Binning Results

It is explained in Section 5.2.2 that for the CAMI Low dataset, we select the contig interval to [750,20000]. Preceding this decision was a series of experiments that were to discover the difference in binning results for various sequence length intervals, and the following section will introduce one of them.

This experiment compares DCMB where the contig interval is set to $[750 - 20000]$ (DCMB: Interval) and DCMB Fixed Size with a fixed sequence length of 750. DCMB Fixed Size cuts all sequences at the length 750, and the remaining part of the sequence is not used. Both models filter out contigs shorter than 750, as their inclusion showed worse binning results.

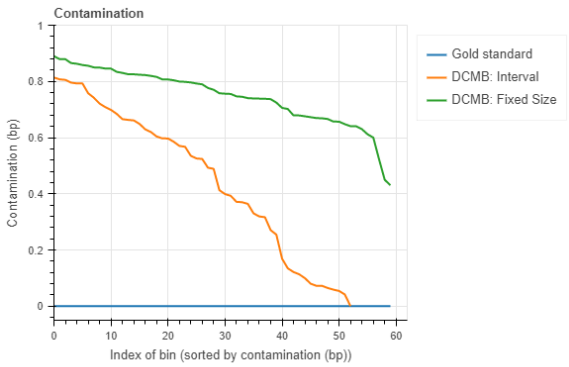


Figure 5.8: Plot representing the quality of bins based on a contamination metric.

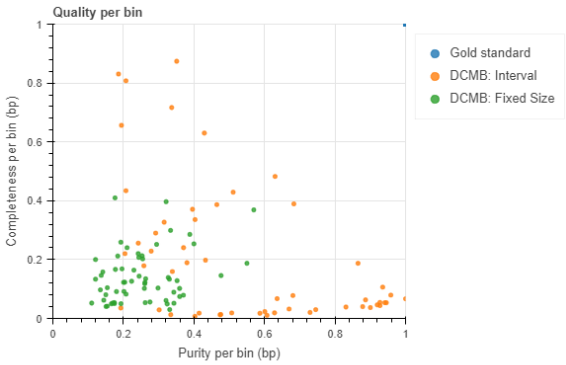


Figure 5.9: Plot representing the quality per bin based on a purity per bin and completeness per bin.

The Figure 5.8 shows a big difference in the level of bins contamination in between the two compared results. While DCMB Interval contains bins that are only slightly contaminated as the curve in the graph slowly descends as expected, DCMB Fixed Size preserves very high contamination across all its bins.

Figure 5.9 uncovers more insights into the results. DCMB Interval manages to create some bins of relatively high completeness and purity. Meanwhile, over 90% of bins belonging to DCMB Fixed Size have low contamination and low purity.

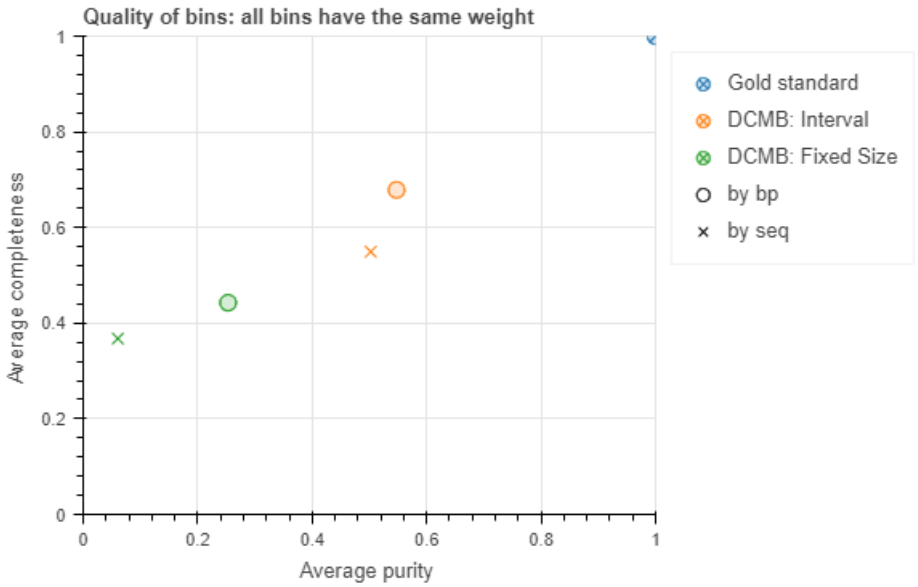


Figure 5.10: Plot representing average quality of bins based on their average completeness and average purity.

Finally, Figure 5.10 confirms the previous findings as the average purity and average completeness of the DCMB Fixed Size bins is falling well behind the DCMB Interval.

In conclusion, cutting the contigs at a fixed length in the lower range of the interval proved not to be a good practice. When we use the full range of sequence lengths, we have to use a lot of padding vectors which might not be optimal for the learning of the model. However, this experimental scenario has demonstrated that the model can learn so much more from the contig interval, than from contigs cut at a fixed length in the lower range of the interval. This means, that the important information that lets the model correctly bin a contig is often not contained within the first 750 nucleotides of the sequence. Longer contigs are the ones that are most often binned correctly by the DCMB model.

5.2.6 Impact of the Number of Clusters on Binning Results

State-of-the-art binners usually manage to create many fewer genomes than there are according to the gold standard. For our experiments, we set the number of clusters to the real number of genomes that are included in the datasets. However, this number is typically not known. This is why the following experiment shows results where we use the single-copy genes method to identify the number of clusters instead of specifying it manually. When using the single-copy genes method,

DCMB discovers 37 bins (DCMB: 37 Clusters). We compare this result with DCMB Best, where the number of clusters is 60 (DCMB: 60 Clusters), and DCMB where the number of clusters is set to 20 (DCMB: 20 Clusters).

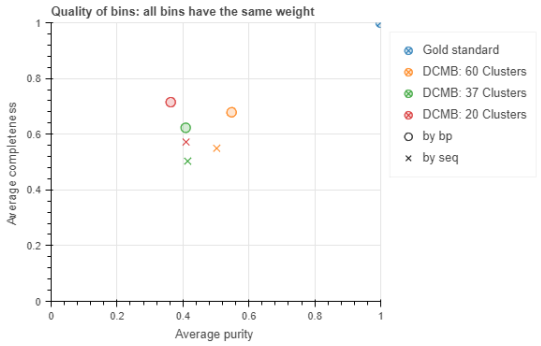


Figure 5.11: Plot representing the quality of bins based on their average purity and average completeness.

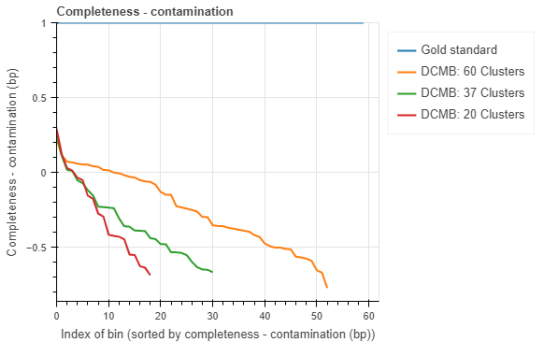


Figure 5.12: Plot representing the quality of bins based on a completeness - contamination metric.

We had an assumption that when dividing the sequences into fewer clusters, the average completeness would be higher, and our results would be closer to the state-of-the-art binners. Looking at Figure 5.11 and Figure 5.12, this doesn't prove to be a trend. The average quality of the bins doesn't show a dramatic difference among the variants. However, DCMB Best still wins by a slight margin. DCMB: 20 Clusters has a slightly higher average completeness but then falls behind in both accuracy and purity. Having fewer bins also results in the bins being more contaminated, which is shown in Figure 5.12.

5.2.7 Discussion and Conclusion

From the result we can see that DCMB is able to learn some meaningful information. However, the signal is not strong enough for DCMB to be used as a stand-alone binner. One of the reasons behind these results might be the larger dimension of the data. In the experiments we have explored different settings, and different types of data preprocessing, as shown in experimental scenario 3. The settings we experimented with included various combinations of loss and activation functions, tuning of convolutional hyperparameters and tuning of parameters of the end-to-end model. However, no settings allowed us to obtain more significant result. Significant results in this context mean to obtain a recovered bin. Our experiments show that it is very hard for convolutional autoencoder to extract meaningful information from raw contigs.

In Section 5.2.5 we have observed that the results tend to improve when using only long contigs of length 20000. However, then there wasn't enough data left

for a proper results, because 95% of sequences are shorter than this value. We can speculate that short contigs don't contain enough information needed for good clusterings.

This trend is the opposite of what we expected. We assumed that the model will learn better from shorter sequences. Due to the limited computational resources, we are limited to six convolutional layers. We expect that this might not be enough to identify the features in large sequences.

As we were unable to recover bins with DCMB, we conclude that the architecture is not fitting for the task of metagenomic binning. We assume that a more complicated architecture might be needed in order to capture the large number of motives that are to be detected in metagenomic data. Nevertheless, there might be opportunities to utilize the DCMB model. One idea could be to use feature embeddings obtained from DCMB as an additional input feature of another binner such as VAMB or DSMB.. This is inspired by the method introduced by Tran et al. [19]. This method uses natural language processing to obtain embeddings, which are later used as an additional input feature. However, the results from DCMB indicate the presence of a lot of noise. This could eventually cancel the positive contribution of such a feature.

In conclusion, the current quality of embeddings is not comparable with the embeddings quality of other binners, namely VAMB, SolidBin and MetaBAT 2. Based on these observations, we will now focus on testing the joint deep clustering method using handcrafted features.

5.3 Deep Stacked Metagenomic Binner (DSMB)

Due to the less significant results of DCMB, it is challenging to explore the effects of the joint deep clustering on this binner. Additionally, the results are hard to compare with the ones of the state-of-the-art binners SolidBin [3], MetaBat2 [2], and VAMB [1]. Therefore, we want to run experiments to explore a stacked-autoencoder-based metagenomic binner that takes composition and abundance as input.

With these experiments, we want to achieve two objectives. First we want to briefly find out how the DSMB approach performs compared to the state-of-the-art binners SolidBin [3], MetaBat2 [2], and VAMB [1]. Thereafter we utilize several experiments to explore the impact of joint deep clustering.

To investigate these objectives, we defined the following experimental scenarios:

1. **Comparison to State-Of-The-Art-Binners on the CAMI Low Dataset**
   This scenario explores how the best DSMB setup performs compared to the state-of-the-art binners SolidBin [3], MetaBat2 [2], and VAMB [1].

2. **Impact of Joint Deep Clustering on the CAMI Low Dataset**
   In this scenario, we compare how the does the best version of DSMB perform compared to a setup that doesn't utilize joint deep clustering and an overfitted version. The setup that doesn't utilize joint deep clustering is utilizing k-means on the learned embeddings in a two-steps approach. In this experiment we want to find out how big the impact of joint deep clustering is.

3. **Impact of Iterations on the Joint Deep Clustering on the CAMI Low Data**
   This experiment is a follow-up experiment to the previous one. We run the DSMB with different number of iterations to investigate the impact of iterations on the binning result.

4. **Impact of Joint Deep Clustering on the Azolla Dataset**
   In this scenario, we explore how the best DSMB setup and the setup that doesn't utilize joint deep clustering perform on the Azolla Dataset compared to the state-of-the-art binners SolidBin [3], MetaBat2 [2], and VAMB [1]. With this scenario, we want to find out if the behavior on a non-synthetic long-read dataset is different compared to the synthetic short-read CAMI Low dataset.

For the settings, it is to mention that we use the Adam optimizer, and the batch size is set to 256. Besides the output layer of the decoder, where we used the sigmoid function as the activation function, the ReLU function is utilized as an activation function. The reconstruction loss is defined by the mean square error

loss. The embedding size is 16. For experiments on the CAMI Low dataset the number of clusters is 60, and for experiments on the Azolla dataset the number of clusters is 18. 60 is chosen, as it is the CAMI Low gold standard. As VAMB recovers 18 clusters on the Azolla dataset, we chose to take the same number. The weight γ of clustering loss L_C is set to 0.1.

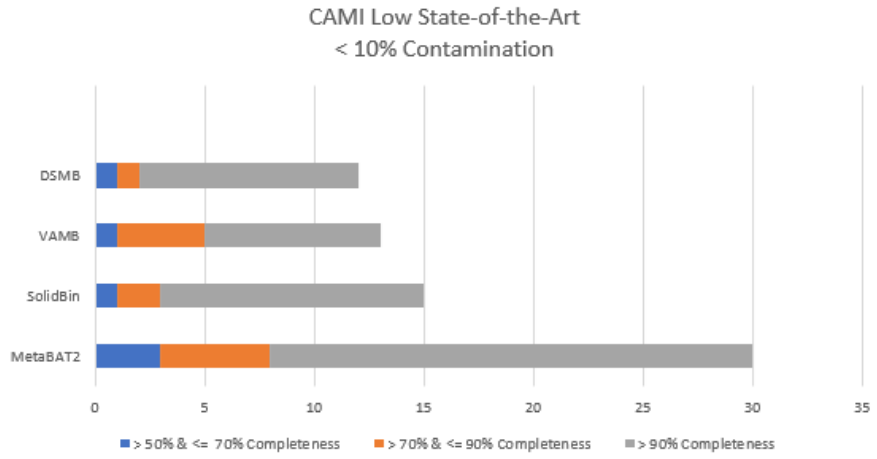
We ran the DSMB with different settings on the CAMI Low dataset and found out that the best parametric settings are with 100 epochs pretraining of the stacked autoencoder and 900 iterations for the entire DSMB. This setting of epochs and iterations is used as the benchmark for the following experiments under the name "DSMB Best". The same settings are also the best settings on the Azolla dataset.

Further to note is that the gold standard of the CAMI Low is 60 bins for recovered bins and 1 for measures like accuracy (bp), average purity (bp), average completeness (bp), and ARI (bp).

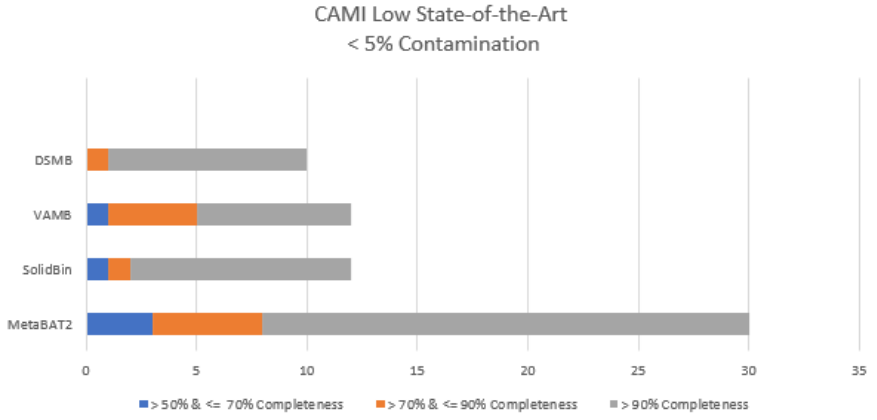
5.3.1 Comparison to State-Of-The-Art-Binners on the CAMI Low Dataset

To find out how the binner performs in comparison with SolidBin, MetaBat2, and VAMB, we plotted the results of the DSMB with the described settings to the results of the CAMI Low benchmark from our previous work [15].

First, we compared the number of recovered bins, as visualized in Figure 5.13.



(a) < 10% Contamination results



(b) < 5% Contamination results

Figure 5.13: Comparison of the number of bins recovered between MetaBat2, SolidBin, VAMB, and DSMB on the CAMI Low dataset.

It is visible that the number of recovered bins of VAMB and DSMB is very close. Given a contamination of < 10%, VAMB recovers 13 bins with a completeness of at least > 50%, while DSMB recovers 12 bins. For the completeness of > 90% it is visible that DSMB recovers 2 more bins than VAMB. DSMB also recovers more bins than VAMB for a contamination level of < 5% and > 90% completeness.

We can conclude that VAMB recovers more bins with lower completeness on the CAMI Low dataset, but DSMB recovers more bins with high completeness. Compared with the insignificant results of the DCMB, DSMB performs well. Comparing the performance of DSMB on the CAMI Low dataset, we can see that DSMB performs in the range of the state-of-the-art binner.

To evaluate the performance of the state-of-the-art binner further, we compare the accuracy (bp), the average purity (bp), the average completeness (bp), and the

adjusted Rand index (bp) of the CAMI Low benchmark. The results are visualized in Table 5.3.

Binner	Accuracy (bp)	Average purity (bp)	Average completeness (bp)	ARI (bp)
MetaBAT2	0.831	0.958	0.581	0.902
SolidBin	0.639	0.638	0.775	0.628
VAMB	0.561	0.824	0.564	0.614
DSMB	0.617	0.767	0.883	0.539

Table 5.3: Comparison of the accuracy (bp), the average purity (bp), the average completeness (bp), and the adjusted rand index (bp) of MetaBat2, SolidBin, VAMB, and DSMB on the CAMI Low dataset.

The comparison shows that DSMB has the third highest purity and the highest overall average completeness. In contrast, the accuracy is the second lowest and the ARI is the lowest. Compared to VAMB, DSMB achieves better results for the accuracy and average completeness. Like VAMB, DSMB is not close to the high results of MetaBAT2.

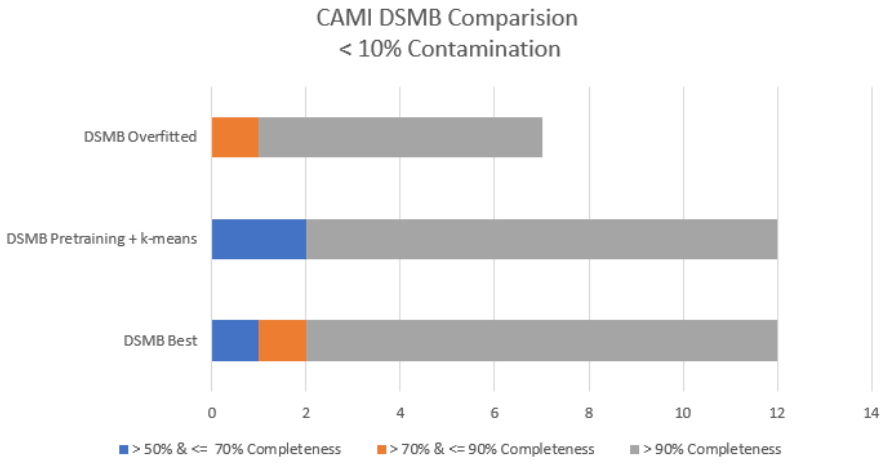
5.3.2 Impact of Joint Deep Clustering on the CAMI Low Dataset

The comparison with the state-of-the-art binners showed that the DSMB performs well. To get an understanding of how much the joint deep clustering (Section 2.7) architecture of the DSMB contributes to the results, we compared the best DSMB settings (DSMB Best) with a two-stage (Section 2.6) setting (DSMB Pretraining + k-means), and a setting that is overfitted (DSMB Overfitted), on the CAMI Low dataset.

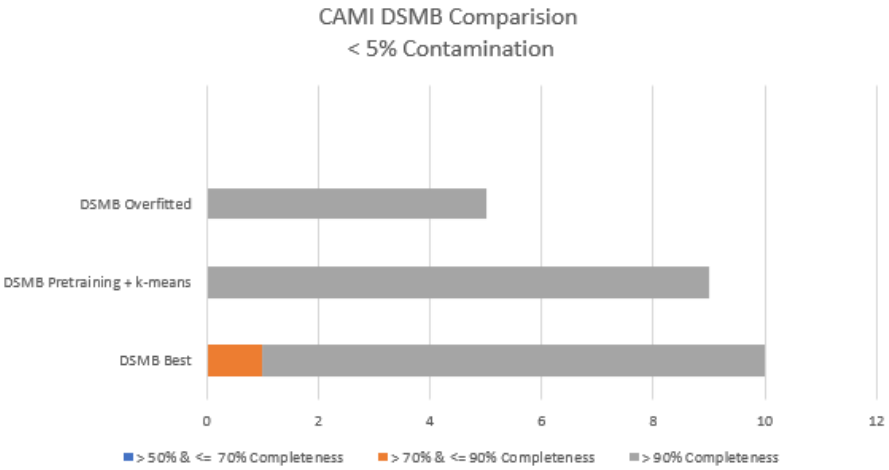
For the DSMB Best and DMSB Overfitted, we learn the representation and clustering jointly. For the overfitted version we run the DSMB with 100 epochs and 11200 iterations. In the context of this experiment, two-stage means that we first learn the representation and then cluster on this embedding.

We created the DSMB Pretraining + k-means by pretraining the stacked autoencoder with 100 epochs, and then we ran k-means on the obtained representations.

For the overfitted setting the number of iterations is set to 11200. The results are illustrated in Figure 5.14.



(a) < 10% Contamination result



(b) < 5% Contamination result

Figure 5.14: Comparison of the recovered bins between the best setting DSMB, the overfitted DSMB and the two-stage DSMB

The results show that DSMB with the best settings performs only slightly better than the approach where it doesn't make use of the joint deep clustering. This means that learning the representation and clusters jointly is just slightly better compared to clustering directly on the learned representations. The difference is only one additional bin for contamination < 5% and completeness from > 70% til < 90% . For a contamination of < 10%, both settings recover the same number of bins. In contrast, DSMB performs far worse than the best setting when overfitted. Here, the number of recovered bins fluctuates between 4 to 5 bins, depending on the completeness and contamination level.

For a more in-depth evaluation, we looked into the completeness - contam-

ination trade-off of this benchmark. This trade-off substracts the contamination from the completeness for each bin. The bins are sorted in descending order. The trade-off is visualized in Figure 5.15.

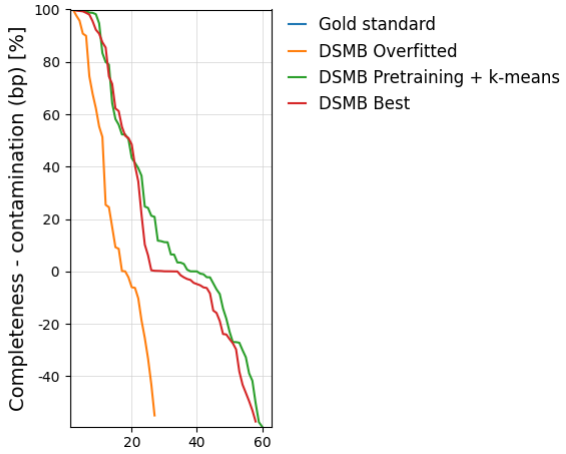


Figure 5.15: Completeness - Contamination trade-off of the DSMB Best Settings, DSMB Two Stage (Pretraining + k-means), and the DSMB Overfitted.

In the plot it is visible that the overfitted DSMB finds fewer 100 % complete bins than the best or two-stage settings. Further, it is visible that the trade-off for the best and two-stage settings are pretty similar.

5.3.3 Impact of Iterations on the CAMI Low Dataset

Since the previous experiment only showed a limited improvement of the clustering results on CAMI Low by using joint deep clustering, it is now in our interest to explore if the results actually gradually improve with increasing number of iterations. To explore this scenario, we ran the DSMB with 100 epochs and a varying number of iterations. The results of the experiment are visualized in Table 5.4 and Table 5.5.

Binner	contamination	> 50 % completeness	> 70 % completeness	> 90 % completeness
DSMB Best 50 Iterations	< 10%	9	8	8
DSMB Best 100 Iterations	< 10%	10	9	8
DSMB Best 400 Iterations	< 10%	9	9	7
DSMB Best 900 Iterations	**< 10%**	**12**	**11**	**10**
DSMB Best 1200 Iterations	< 10%	10	10	10

Table 5.4: Comparison of the number of bins recovered for DSMB with 100 epochs and a varying number of iterations from 50 to 1200. Contamination of 10%.

Binner	contamination	> 50 % completeness	> 70 % completeness	> 90 % completeness
DSMB Best 50 Iterations	< 5%	9	8	8
DSMB Best 100 Iterations	< 5%	7	7	7
DSMB Best 400 Iterations	< 5%	6	6	4
DSMB Best 900 Iterations	**< 5%**	**10**	**10**	**9**
DSMB Best 1200 Iterations	< 5%	8	8	8

Table 5.5: Comparison of the number of bins recovered for DSMB with 100 epochs and a varying number of iterations from 50 to 1200. Contamination of 5%.

The tables show that the number of recovered bins is getting bigger for a contamination of < 10% when 100 iterations is reached. Then the number of recovered bins decreases, just to be improving at 900 iterations, where the maximum number of bins was recovered. Thereafter the number of recovered bins declines again. From the table it can be concluded that considering the number of recovered bins there is a local maximum at 100 iterations and a global maximum at 900 iterations.

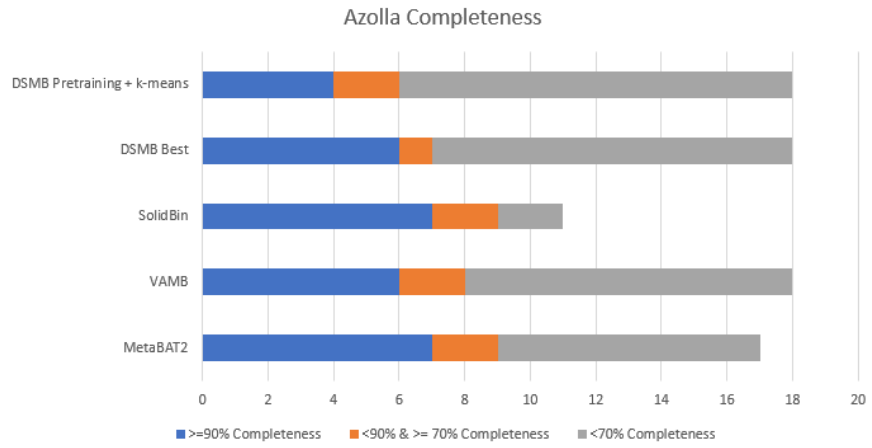
That means that the joint deep clustering is not steadily improving the number of recovered bins. Comparing the results for a contamination of < 5%, slightly fewer bins are recovered after 400 iterations than there are after 50 iterations.

5.3.4 Impact of Joint Deep Clustering on the Azolla Dataset

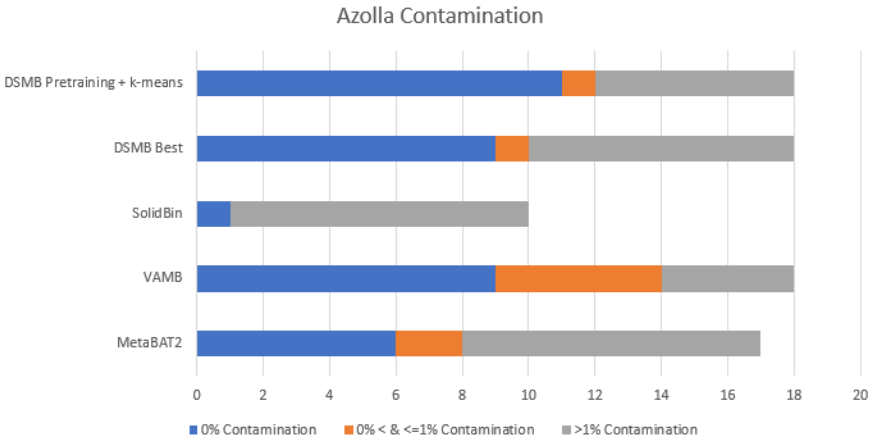
So far, we performed experiments on the synthetic short read CAMI Low dataset. To evaluate the performance for non-synthetic long-read data, we ran DSMB with the best and two-stage setting on the Azolla dataset.

The best settings for the Azolla dataset are found with 100 epochs and 900 iterations, and this is therefore similar to the best settings for the CAMI Low dataset. To compare the results with the state-of-the-art binners SolidBin [3], MetaBat2 [2], and VAMB [1] we plotted the DSMB results together with the results of our previous work [15].

The results for completeness and contamination are illustrated in Figure 5.16.



(a) Completeness results



(b) Contamination results

Figure 5.16: Completeness and Contamination results for the Azolla dataset.

Regarding completeness, DSMB Pretraining + k-means and DSMB Best find the smallest number of bins that are over 70% complete. For completeness between 70 and 90%, VAMB recovers 1 bin more than DSMB Best. Looking at results for completeness over 90%, DSMB Best recovers 2 bins more than the two-stage (Pretraining + k-means) DSMB.

Regarding contamination, the only binner that has less contaminated bins than DSMB is VAMB. The two-stage DSMB recovers 2 more bins with 0% contamination than DSMB Best.

The reason for that behavior is that the two-stage DSMB recovers more bins with less completeness but also less contamination, compared to DSMB Best.

Chapter 6

Discussion

In this chapter, we will discuss different aspects of the project and report. We will present an overview of both metagenomic binners that we created throughout the project. Next, we will list the main challenges we faced and sum up the main findings from our experiments. Lastly, we will provide an explanation behind the main discoveries and outline some suggestions for further work.

6.1 Discussion: Deep Convolutional Metagenomic Binner (DCMB)

While we were creating a metagenomic binner that will take raw contigs as its inputs, we discovered two main challenges. One challenge is to design a deep neural network that can effectively learn feature representations from our sequential data. The second challenge is to find such data preprocessing steps that will lead to the best clustering results. The clustering algorithm itself does not need any special adjustments for clustering on sequential data embeddings. The success of clustering lies mainly in constructing a network that will compress the input data into meaningful representations.

We can be more specific and present some of the bigger challenges that we faced. Firstly, suitable data preprocessing has a major effect on the final results. We had to find a way to handle varying length of the sequences (contigs). This was solved by selecting an interval of the sequences lengths. This interval would be specific to every dataset. The reason is that this interval should cover the most common contig length after filtering out short contigs. For CAMI Low, this interval was [750, 20000]. Next, the sequences were one-hot encoded. The encodings shorter than the maximum length were padded with zeros.

In general, the problem with using raw contigs as an input feature is their length. If we wouldn't select a maximum length at which we cut the remaining

part of the contig, we would have a huge input data dimension. The cause is that the longest contigs, even in a synthetic CAMI Low dataset, contain a few million nucleotides. Processing such a length would require enormous computational resources. Additionately, the ratio of 0s to 1s would further increase due to the required padding. Further, this would negatively affect the learning capability of the model. It would be too difficult for the model to learn meaningful feature representations.

For comparison, the dimension of CAMI Low after cutting sequences at the length 20000 is $20000 \times 4 = 80000$. The dimension of MNIST dataset is $28 \times 28 = 784$.

Secondly, we found out that it is difficult for an autoencoder to reconstruct one-hot encoded inputs. Our experiments showed that the decoder tends to reconstruct all inputs in about the same value (such as 0.5), instead of either 0 or 1. This was partially solved by using a suitable loss function, namely cosine embedding loss.

6.1.1 General Results and Observations

Our experiments showed, that DCMB has some learning capability. However, it is not enough to recover any genome. In our best run, we achieved the model's accuracy of 0.33. Even though a third of the contigs are clustered correctly, the clusterings also contain a lot of noise. That compromises the positivity of the result. The current quality of the embeddings is not comparable with the quality of other binners, namely VAMB, MetaBAT2, SolidBin, and also our second binner DSMB.

We still think that feature representations obtained from raw contigs could be used as an alternative to the composition (TNF) input feature. It is likely that feature representations from raw contigs don't have potential to replace the abundance feature. However, they could be used in combination with the abundance as an input to another model.

All in all, we attempted to learn feature representations from raw contigs, and it turned out that the problem is very difficult. That is due to the length of sequences and the overall hardness of the problem. These make it uneasy for a model to learn meaningful embeddings.

We have learned that the DCMB model has more difficulty clustering correctly short contigs than it has when clustering longer contigs. We achieved a higher clustering accuracy when we filtered out contigs shorter than 750. We observe this because short contigs can fit well in many genomes. They don't contain enough information to be clustered correctly.

On the other hand, large contigs perhaps include enough information. However, our computational resources didn't allow us to scale the architecture of the model to the point where it would be able to effectively abstract all important

features.

It is possible that the architecture of our autoencoder wasn't optimal for the task. Some ideas for future work include composing a more complicated CNN-based autoencoder with more encoding and decoding parts. It would also be interesting to experiment with Long short-term memory (LSTM) autoencoder architecture or combining the two mentioned.

6.1.2 Impacts of Joint Deep Clustering on DCMB

The experiments showed that DCMB benefits from the deep joint clustering method. This was demonstrated in experiment 5.2.4. The difference in the results is not drastic, and the improvement is more like fine-tuning. However, we do observe slightly better results by employing this technique. Even though the metrics registered some improvement with joint deep clustering, the results were not significant. The method didn't help to recover any genomes. In order to investigate the impact of the joint deep clustering further, we created a second binner, Deep Stacked Metagenomic Binner (DSMB) and focused on clustering hand crafted features.

6.2 Discussion: Deep Stacked Metagenomic Binner (DSMB)

Based on the difficulties we had when using raw contigs as input, we decided to focus on the hand crafted features, namely composition and abundance. We assumed that we could achieve significant results with a suitable autoencoder architecture. The reason is that those features are proven to be a sufficient source of information for successful clustering results by other binners. That would let us study the effect of the deep joint clustering with more certainty.

DSMB differs from DCMB mainly in the type of autoencoder network. Here, we used stacked autoencoder with fully connected layers. The main challenge from the implementation point of view was correct computation and normalization of the abundance and composition features. It was also important to determine the optimal number of encoding and decoding layers. However, the configuration of the stacked autoencoder was less difficult than it was for the convolutional autoencoder in DCMB. DSMB requires fewer hyperparameters to tune. The joint optimization of feature representations and clustering remained the same as in DCMB. That is why we explained DSMB more briefly, even though it has at least equal importance in the project.

The assumption about the bigger significance of the results when using traditional features has proven to be correct. In general, the results are comparable to the state of the art results, especially VAMB. Therefore, hand crafted features are a better source of information than unprocessed contigs.

6.2.1 Impacts of Joint Deep Clustering on DSMB

Through our experiments in Section 5.3.2 and Section 5.3.3 we observed a slight improvement when jointly optimizing feature representations and clusters. This is opposed to the two-stage method where we ran a clustering algorithm k-means on feature representations created in pretraining.

Similarly to what we observed with DCMB, the improvement is again more like fine-tuning. Most of the experiments managed to recover between 1 to 2 more bins when using the joint method. There is a high probability of recovering more bins when adding this optimization step, however, that is not guaranteed. This is not an unusual observation for a fine-tuning optimization technique: it does not always have a significant impact, especially if the weights of the model were already well-adjusted from the pretraining step.

We have observed that with metagenomic data the model is prone to overfitting. This is not something that was reported for the original DCEC model [67]. To mention is that the DCEC run with image data. Due to this behavior, we have found the stopping criterion originally proposed for DCEC ineffective. The original stopping criterion is looking at the percentage of embedded points that were reassigned to a different cluster in between two consequent iterations. Even after adjusting the threshold tolerance to a higher number, the stopping criterion has proven to be unreliable. This is why we decided to use an early stopping criterion instead.

Updating Target Distribution The experiment in Section 5.3.3 has shown that the improvement of clustering results is not continuous as we would expect. We observed that the number of recovered genomes, as well as other measures, repeatedly improves and worsens again throughout the iterations. We discovered that this behavior might be affected by updating the auxiliary target distribution. The original DCEC method recalculates this distribution after every 140 iterations. Even though this updating interval is configurable, we didn't observe any benefits of this action for metagenomic data. The left graph in Figure 6.1 shows that when the target distribution is updated, the clustering loss L_c jumps up, to partially decrease again throughout the following iterations. It is expected to observe the clustering loss to get bigger when the target distribution is updated. However here, the total loss L gets gradually bigger. This is not something we should be observing. The loss L is the sum of the clustering loss L_c and the reconstruction loss L_r:

$$L = 0.9 \times L_r + 0.1 \times L_c \tag{6.1}$$

The right graph in Figure 6.1 shows the development of loss functions without target distribution updating. In this case, the loss smoothly converges.

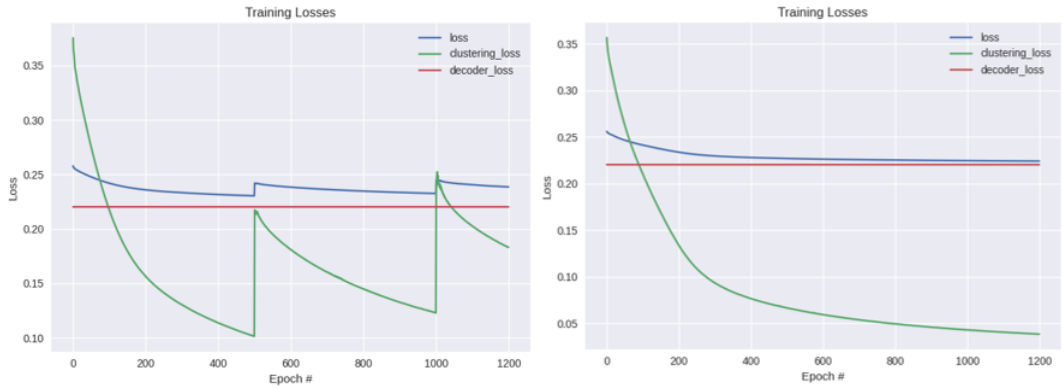


Figure 6.1: Trends of loss functions during the training process. The left graph shows the loss trends for DCMB, where target distribution updates after each 500 training iterations. The right graph shows the loss trends for DCMB model where target distribution does not update.

Figure 6.2 shows that recalculating the target distribution does not provide any observable benefits to the training process. Moreover, the setup without the target distribution update achieves slightly better results overall.

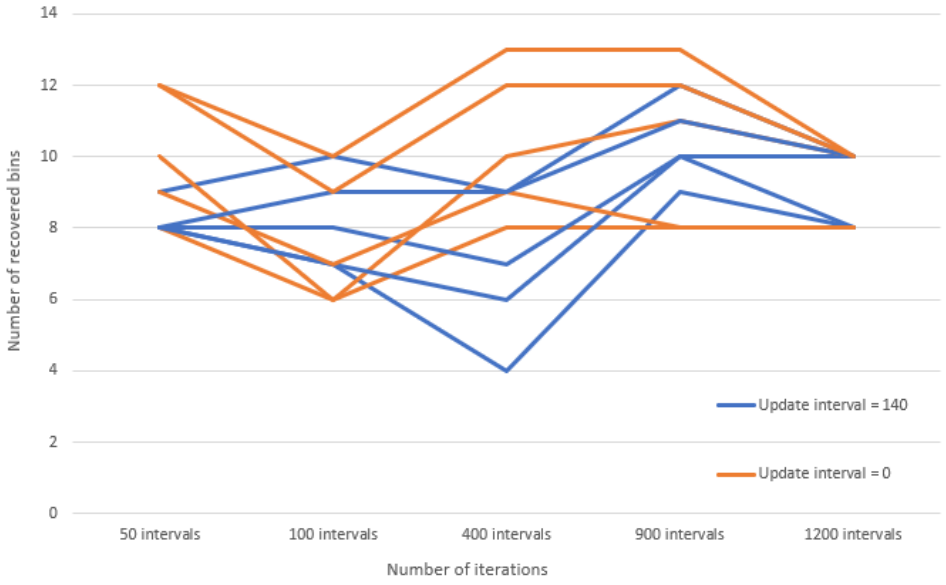


Figure 6.2: Trend of the number of recovered genomes throughout the model training. The graph shows the number of recovered genomes after varying the number of training iterations. It compares a setup where the target distribution updates every 140 intervals (blue lines), and a setup where the target distribution doesn't update (orange lines). Each line connects results for a specific contamination level (<5% or <10%), and completeness level (>50%, >70%, >90%).

Therefore for metagenomic binning, we would recommend only calculating the

target distribution one time during the 0th iteration and then minimizing the KL divergence between soft cluster assignments and this initial target distribution.

At the moment, the number of clusters has to be estimated in a separate step. Future work would include incorporating the number of clusters estimation into the end-to-end solution.

In conclusion, the deep joint clustering method that we explored in this project is not computationally expensive when using handcrafted features, and most of the time, it leads to improved results. The main disadvantage of this method is its tendency to overfit metagenomic data. If this problem is resolved, the joint deep clustering using the KL divergence clustering technique will be a viable optimization technique for improving binning results.

Chapter 7

Conclusion

In this chapter, we reason how the content of the report answers the questions formulated in the problem statement in Section 1.2.

The report first explains the fundamentals required for answering the problem statement questions. This is why we introduced the field of (meta)genomics, described the structure of DNA, and explained the sequencing process (Section 2.1). Since our problem definition requires an understanding of the deep learning field, we have also explained the basic idea behind deep neural networks and the building blocks that such networks consist of (Section 2.2).

To find the answers to our problem statement questions, we have created and evaluated two new metagenomic binners: Deep Convolutional Metagenomic Binner (DCMB) and Deep Stacked Metagenomic Binner (DSMB). Both of the binners consist of an autoencoder part and a deep clustering part. Each of the binners was designed with a specific purpose. With DCMB, we explored the potential of raw contigs as input features. With DSMB, we studied the impacts of joint deep clustering on the number of recovered genomes. The ultimate goal was to explore innovative solutions that could be useful for improving the metagenomic binning results of existing and future binners.

The problem statement itself consists of two questions. The first question in the problem statement asks:

- Can we benefit from the sequential information in contigs and achieve competitive binning results by implementing a deep learning model that will use them as an input?

Hand-crafted input features used by many binners are extracted from raw DNA sequences in an additional preprocessing step. We explored if it is feasible to skip this step by using raw contigs as inputs. Such a solution eliminates the need for additional preprocessing and makes use of sequential information that is found in raw contigs. To answer the question, we created Deep Convolutional Metagenomic

Binner (DCMB). We have discovered that feature representations can be learned from sequential data with a convolutional autoencoder. We explained the architecture of this metagenomic binner and why we think the presented architecture is suitable for our problem. One of our main challenges was adapting a convolutional autoencoder for sequential input data. Typically, CNN-based autoencoders are trained with image datasets. Throughout the first part of Chapter 5, we documented our experimental setup and scenarios that explore the potential of using unprocessed contigs as input features in our DCMB model. These experiments showed that the feature representations obtained from raw contigs alone are now powerful enough, and the binner does not achieve competitive results when clustering on them. Finally, the problem statement question is answered throughout Section 5.2.7 and Chapter 6, Section 6.1.

With our current implementation of DCMB, we can not benefit from raw contigs being the only input feature of a stand-alone binner. Nevertheless, we have gained interesting insights into the problem area. This includes detection of certain learning abilities of such binner. We believe that with different architecture and sufficient computational resources, utilizing raw contigs in a metagenomic binner might still bear potential.

The second question formulated in the problem statement asks:

- Can we benefit from involving clustering deeper during the training phase?

To answer this question, we first needed to get a deeper understanding of the problem area. This is why throughout Chapter 2, we explained deep clustering methods and clustering techniques commonly used in those methods. In conclusion, we decided to focus specifically on joint deep clustering methods. Those are methods that involve clustering within the training process, as they optimize feature representations and clustering simultaneously. We described selected joint clustering frameworks and analysed their clustering techniques. We decided to use a joint clustering method that utilizes KL divergence clustering technique. This method initializes clusters on learned embeddings with k-means and then refines them using the KL divergence. Currently, this method is used by a few existing frameworks, including DCEC and DEC.

We experimented with this joint deep clustering method in our DCMB binner, where the experiments showed improvement in some metrics. The less significiant DCMB results made it uneasy to observe the effects of joint clustering. To investigate the effects of joint clustering better, we implemented a second binner called Deep Stacked Metagenomic Binner (DSMB). This binner takes abundance and composition as input features, and it recovers a competitive number of genomes. We explained the architecture of this binner and the reasoning behind it. After evaluating a series of experiments in Chapter 5, we answered the problem statement question in Chapter 6 Section 6.2.

In summary, we have discovered that in most cases, we do benefit from the utilization of the joint deep clustering method in metagenomic binning. We showed that by using this method, we can optimize the learned feature representations and recover a slightly larger number of genomes.

Future work would include integration of the number of the bins estimation and improved overfitting prevention.

Appendix A

Apendix

A.1 Previous Experiments

The following section describes the findings of the group's previous report [15].

One of the aims in the previous report was to explore how the binning results of VAMB compare to the state-of-the-art binners MetaBAT2 [2] and SolidBin [3].

MetaBAT2 utilizes probability-based abundance and composition scores and then performs an LPA (label propagation algorithm) for clustering. SolidBin performs a type of spectral clustering with incorporated pairwise constraints. To compare the performances, a metagenomic binning benchmark was conducted on the datasets CAMI low [4], Azolla [7] and Strong100 [101]. To access the quality of the resulting bins, a tool AMBER [102] (Assessment of Metagenome BinnERs) was used for the Strong100 and Azolla dataset, and a method CheckM [103] for the Cami low dataset. The average size of the datasets was around 15 GB. Figures and tables with the results of the benchmark can be found in Appendix A.2 of the Appendix. Briefly, the results of the benchmark showed, that VAMB trades off low completeness for low contamination, and SolidBin trades off high completeness for high contamination. MetaBAT2 showed the most consistent results without trading off completeness and contamination.

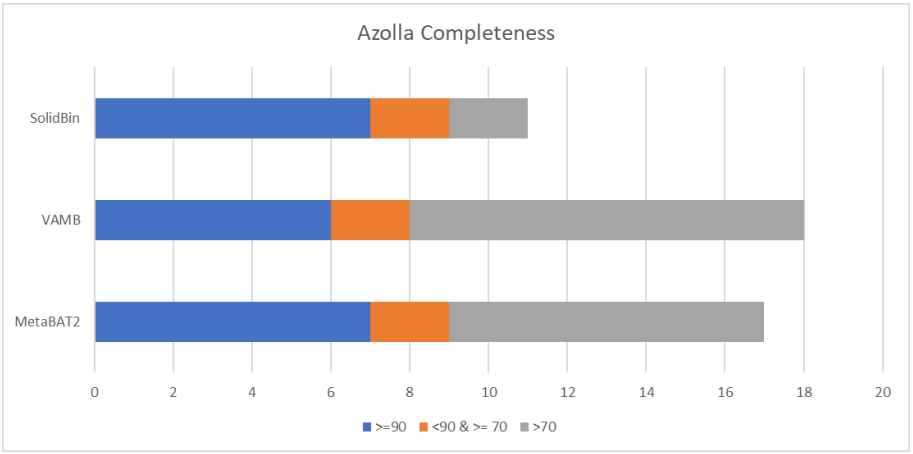
A.2 Metagenomic Binner Benchmark

This section provides the results of the benchmark with the binners SolidBin, MetaBAT2 and VAMB of our previous report [15].

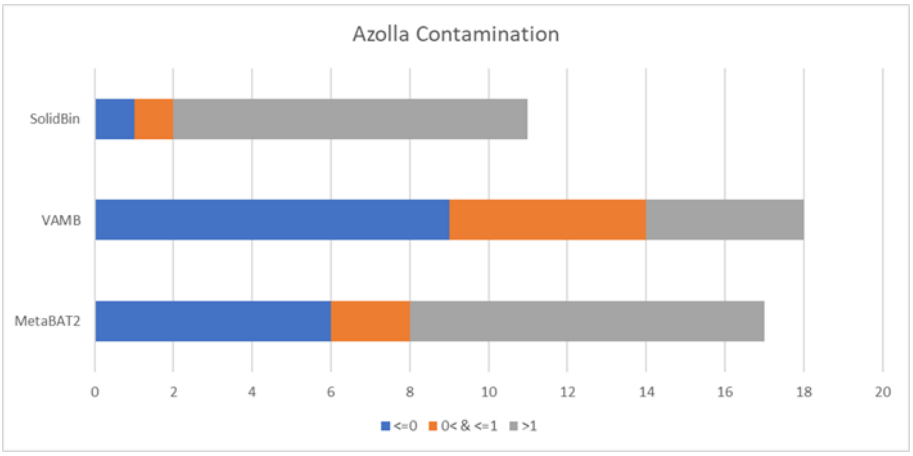
Dataset	VAMB	MetaBAT2	SolidBin
Azolla	18	17	11
Strong 100	60	69	42
CAMI Low	13	30	15

Table A.1: Recovered bins for every binner and dataset.

A.2.1 Azolla Dataset



(a) Completeness results

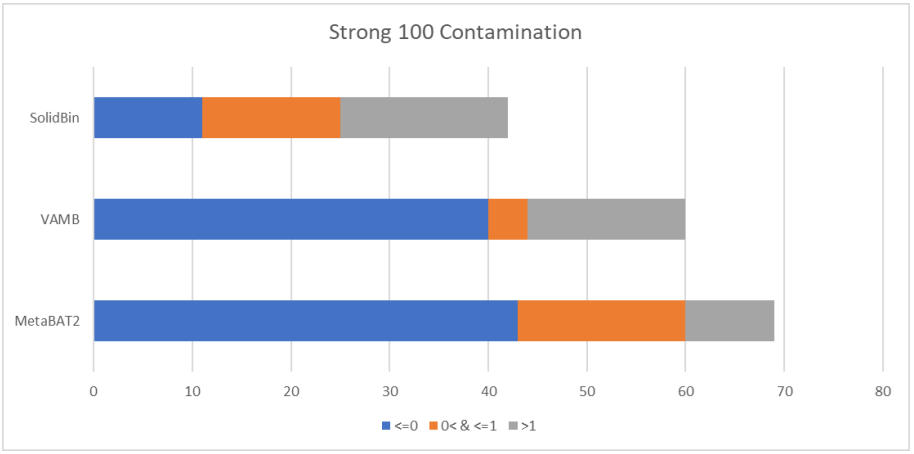


(b) Contamination results

Figure A.1: Results of the benchmark performed on the Azolla dataset.

A.2.2 Strong 100 Dataset

(a) Completeness results



(b) Contamination results

Figure A.2: Results of the benchmark performed on the Strong 100 dataset.

A.2.3 CAMI Low Dataset

Binner	contamination	> 50 % completeness	> 70 % completeness	> 90 % completeness
Gold Standard	< 10%	60	60	60
Gold Standard	< 5%	60	60	60
MetaBAT2	< 10%	30	27	22
MetaBAT2	< 5%	30	27	22
SolidBin	< 10%	15	13	12
SolidBin	< 5%	12	11	10
VAMB	< 10%	13	12	8
VAMB	< 5%	12	11	7

Table A.2: Completeness and Contamination of the CAMI Low dataset

A.3 DCMB additional Experiments

A.3.1 Base Case Versus Result From Short Training.

The purpose of this experiment is to show if there is an improvement in results between the improperly trained DCMB model (DCMB Short Training) and DCMB Best. This is to explore the learning power of the properly trained DCMB Best model.

Comparing our best result with the result from short training in Figure A.3, it can be noted that the model can learn partially correct information throughout the training, even though the quality of learned features is not sufficient for a competitive result. Both experiments were run with the number of clusters set to 60. DCMB Best ended with 52 clusters, while the DCMB Short Training model was only able to identify 12 clusters. This indicates that DCMB Best learns enough information to identify the requested number of clusters.

Figure A.3 shows the quality of bins taking both completeness and contamination into account, while Figure A.4 shows the quality of bins when only looking at how much contaminated they are. In both graphs, the properly trained DCMB model produced bins of strongly better qualities.

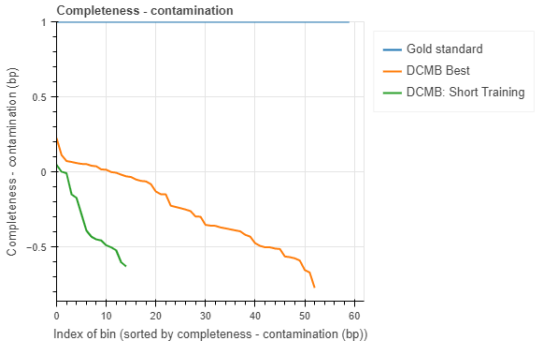


Figure A.3: Plot representing the quality of bins based on a completeness - contamination metric.

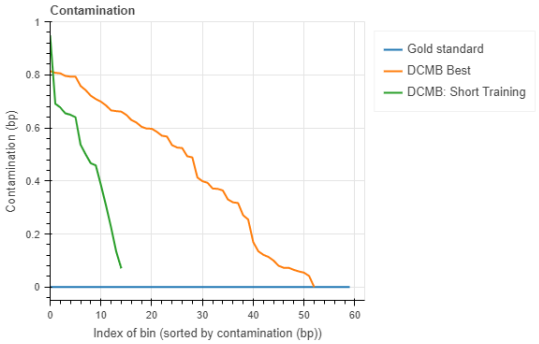


Figure A.4: Plot representing the quality of bins based on a contamination metric.

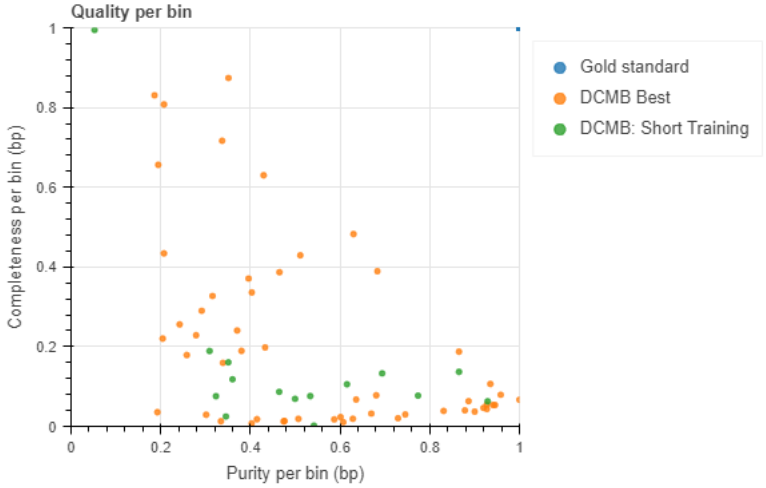


Figure A.5: Plot representing the quality of bins based on purity and completeness for each bin.

Figure A.8 illustrates the quality of each bin, considering their completeness and contamination. The DCMB Best contains 3 bins that are more than 40% complete and more than 40% pure, and it contains 19 bins that are more than 20% pure and more than 20% complete. Meanwhile, DCMB Short Training contains a single bin that is 100% complete and extremely highly contaminated, and the rest of the bins are only around 10% complete, with 2 bins being more than 80% pure. This indicates that DCMB Short Training contains 1 huge bins and a couple of very small bins.

Binner	Accuracy (bp)	Av. purity (bp)	Av. completeness (bp)	ARI (bp)
Gold Standard	1	1	1	1
DCMB Best	0.33363	0.5475	0.67861	0.22901
DCMB Short Training	0.09066	0.51048	0.91841	0.00391

Table A.3: Comparison of accuracy (bp), average purity (bp), average completeness (bp), and adjusted Rand index (bp) between DCMB Best result, and DCMB Short training.

Table A.3 provides an overview of the basic metrics for both compared models. DCMB Best achieves a decent base-pair accuracy of 0.33 and adjusted Rand index of 0.23. Although a lot of noise is present in the results, the presented metrics indicate some success in the binning process. In comparison with the poor results of DCMB Short Training, we can state that the results are not accidental, and DCMB demonstrates a learning potential.

A.4 DCMB Structure

The following figure is additional material that shows the implementation structure of DCMB.

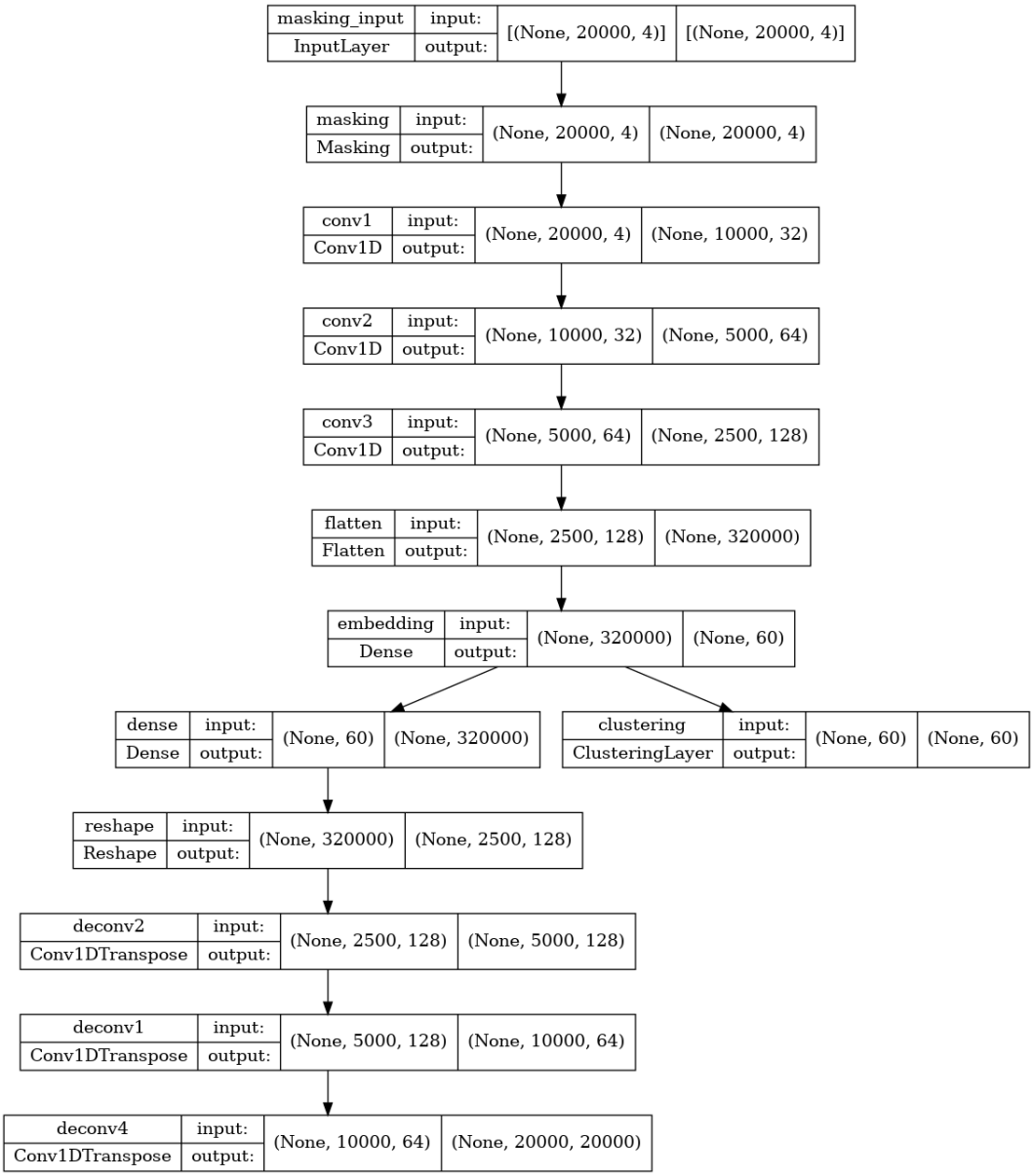


Figure A.6: Detailed illustration of the DCMB structure with attached clustering layer.

The encoder takes encoded raw contigs cut to the size of 20000 base pairs as input. Through 3 convolution layers, embeddings of size 60 are obtained. The embeddings are input to the decoder and the clustering layer.

The decoder takes embeddings of size 60 as input, and outputs encoded vectors of size 20000. Bins are obtained through the clustering layer.

A.5 DSMB Autoencoder Structure

The following figures are an additional material that shows the implementation structure of the encoder and the decoder of DSMB.

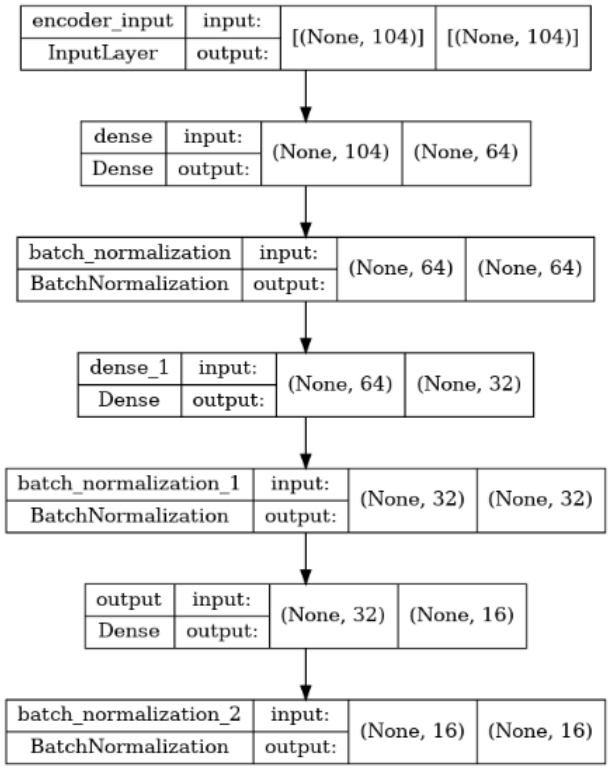


Figure A.7: Detailed illustration of the DSMB encoder.

The encoder takes concatenated abundance and composition vectors as input. Each of these vectors has a size of 104. Through 3 dense layers, embeddings of size 16 are obtained.

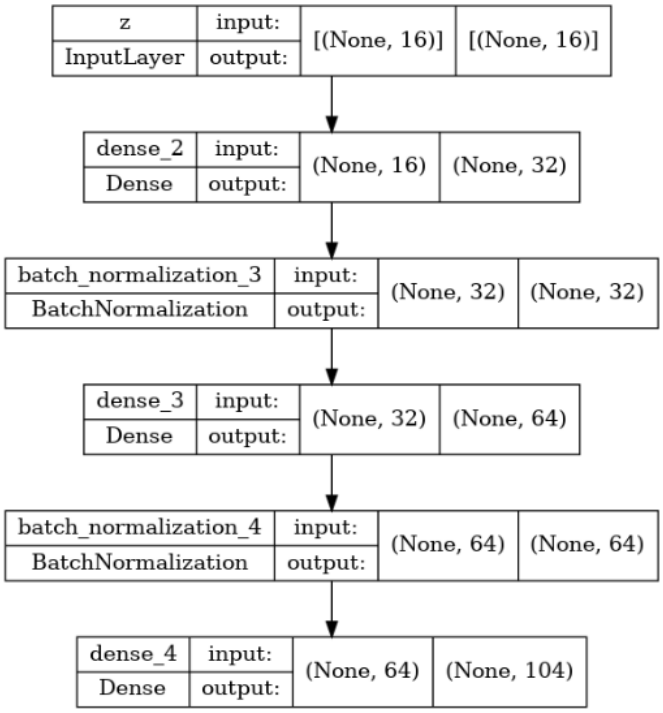


Figure A.8: Detailed illustration of the DSMB decoder.

The decoder takes the embeddings of size 16 as input. The outputs are concatenated abundance and composition vectors of size 104.

A.6 DVMB - Deep Variational Metagnomic Binner

A.6.1 Idea

The main idea of this model is to apply and verify a deep metagenomic architecture in a joint learning context. We have decided to base this approach on the DCEC and to use a variational autoencoder as a network structure. We utilize a variational autoencoder because we want to use a network similar to VAMB.

A.6.2 Architecture

The overall DVMB structure (Figure A.13) is based on the DCEC model (Section 2.8.3) in which we replace the convolutional autoencoder (CAE) by the variational autoencoder (VAE). Similarly to the DSMB, DVMB takes TNF and abundance as its input (Section 5.3) and consists of two main parts:

- The variational autoencoder for constructing feature representations from the input data (TNF and Abundance)

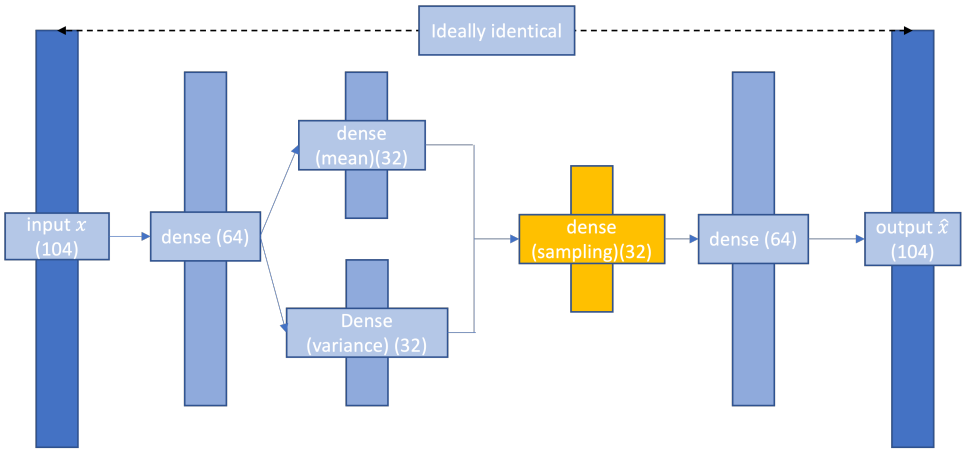


Figure A.9: The overall structure of the VAE used for the DVMB.

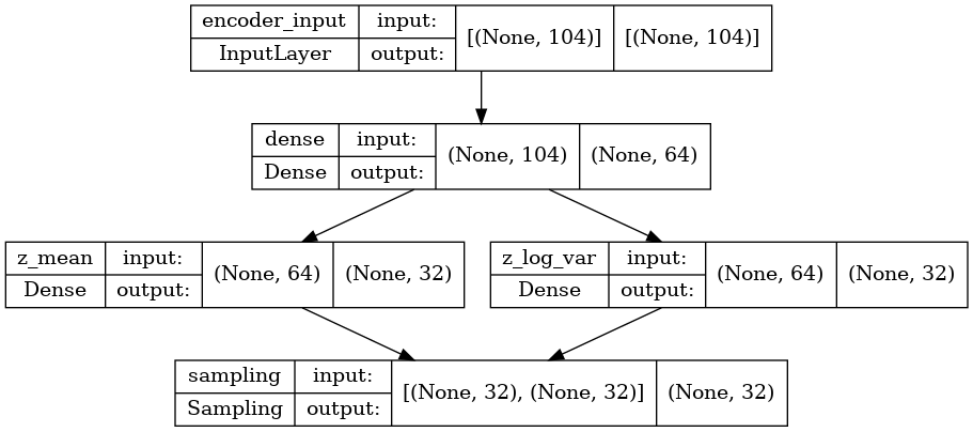


Figure A.10: Detailed illustration of the encoder.

The *encoder_input* layer has 104 input neurons and processes the concatenation of TNF and abundance vectors. The outputs will be passed into a *dense* layer which has 64 hidden neurons where embeddings are generated. The two embeddings (*z_mean* and *z_log_variant*) are organized in two separated 32-neurons *dense* layers. Finally, two embedding layers are sampled into the *sampling* layer.

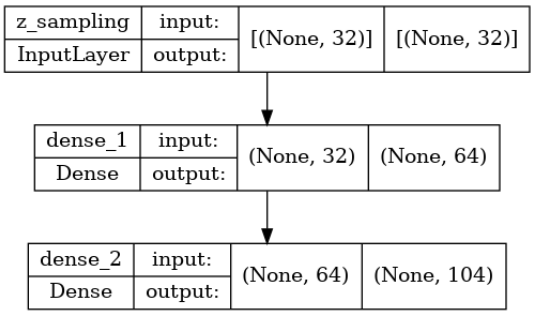


Figure A.11: Detailed illustration of the decoder.

The *decoder* is connected to *sampling* layer and use its input to reconstruct *z_mean* and *z_log_variant* embeddings. It enhances the encoder's performance through backpropagation process.

- The clustering layer Section 2.8.6 accepts input from the most data-rich parts of the latent space (μ (mean)) and optimizes the predicted soft labels q for all data point over the iterations. The following figures will demonstrate how the clustering layer is integrated into the DVMB's overall structure.

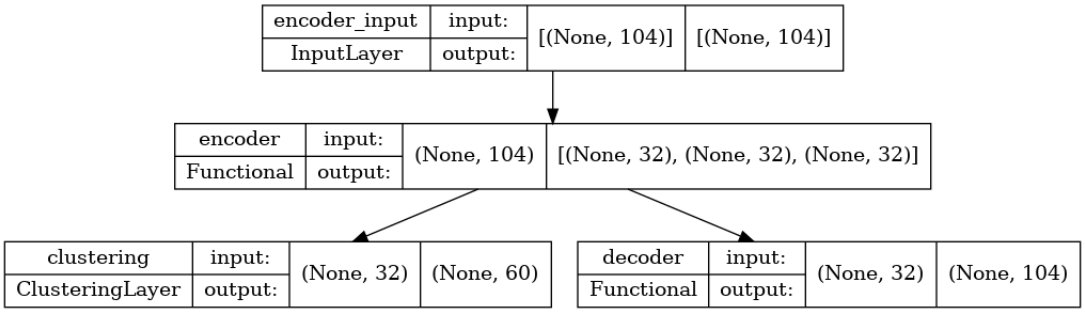


Figure A.12: Illustration of DVMB model which performs end-to-end learning.

By combining the clustering layer and the VAE, we form the DVMB joint clustering method:

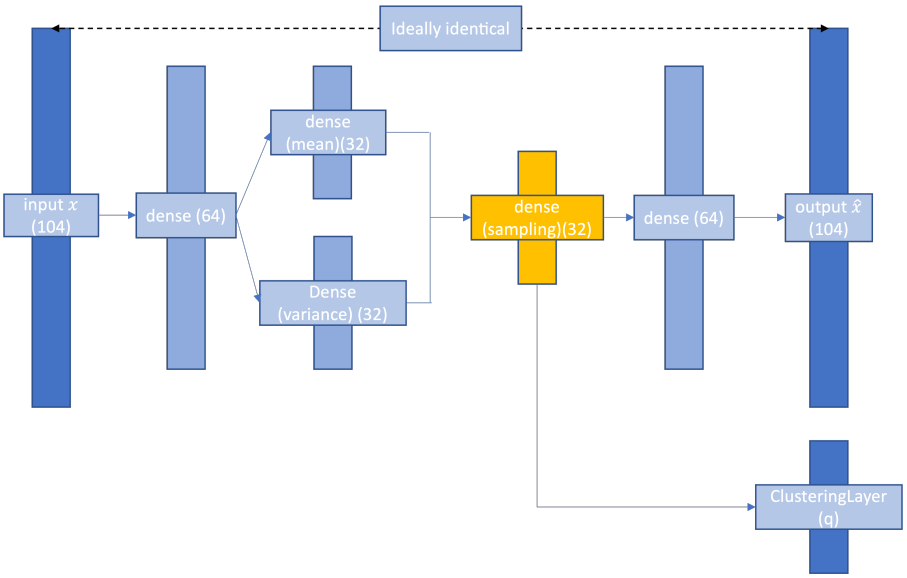


Figure A.13: The overall overview of DVMB.

A.6.3 Evaluation

We validated the DSMB against the DVMB on the CAMI Low dataset. The CAMI Low dataset was chosen as we wanted to take advantage of the supported ground truth for a more accurate evaluation. This section will explain how we evaluate the DVMB against the DSMB. Mainly, we want to compare the contribution of the joint clustering to the results. For this experiment, we used mostly the same settings for the DSMB and DVMB as described in section 5.3. For each run, we changed the number of trained iterations. For consistency with other parts in this report, we select the three train iterations: 100, 900, and 1200 for this experiment. To validate the DVMB's joint clustering performance in comparison with the normal two-stage training, we execute one additional two-stage "Pretraining + k-mean" experiment where we pretrain DVMB's variational autoencoder in 100 epochs and then perform k-means on the feature representation which obtained after retrain.

Result and observations

#	Tool	Contamination	> 50% Completeness	> 70% Completeness	> 90% Completeness
0	DVMB 1 Iter	10%	6	6	6
1	DVMB 1 Iter	5%	5	5	5
2	DVMB 100 Iter	10%	4	4	4
3	DVMB 100 Iter	5%	3	3	3
4	DVMB 900 Iter	10%	6	5	5
5	DVMB 900 Iter	5%	5	4	4
6	DVMB 1200 Iter	10%	7	6	6
7	DVMB 1200 Iter	5%	4	3	3

Table A.4: DVMB results.

#	Tool	Contamination	> 50% Completeness	> 70% Completeness	> 90% Completeness
1	DSMB 100 Iter	10%	14	13	13
2	DSMB 100 Iter	5%	12	11	11
3	DSMB 900 Iter	10%	7	7	6
4	DSMB 900 Iter	5%	7	7	6
5	DSMB 1200 Iter	10%	11	11	9
6	DSMB 1200 Iter	5%	10	10	9

Table A.5: DSMB results.

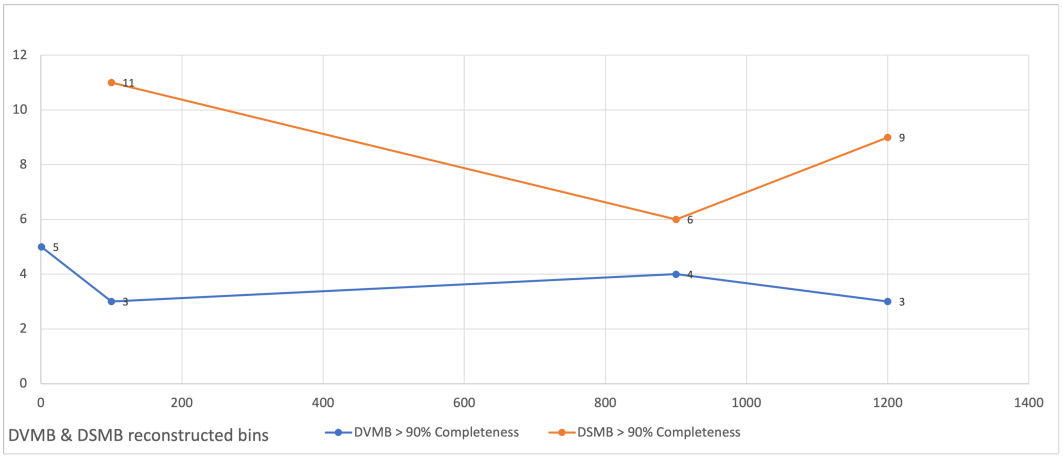


Figure A.14: DVMB & DSMB 5% Contamination comparison.

The comparison in Figure A.14 extracts bin reconstruction results from the highest quality criteria. From the graph in Figure A.14 we can conclude the following:

- The DSMB model is outperforming the DVMB in all of the experiments. DSMB creates more qualified bins (double the number of high quality bins after 100 iterations) on the CAMI Low dataset [4] using the AMBER tool [102]. This might be due to the fact that the VAE part of the DVMB involves two separated embeddings (mean and variance), and with additional interfere from the clustering layer, we need a more sophisticated mechanism than the one proposed by the DCEC.
- The joint structure in this case does not show significant improvements when compared with the "Pretraining + k-means" setting. We can only see small improvement at 1200 train iterations.

A.7 Implementation Source

- DCMB: https://github.com/leotimus/DMB/tree/dcmb
- DSMB: https://github.com/leotimus/DMB/tree/dsmb
- DVMB: https://github.com/leotimus/DMB/tree/dvmb

List of Acronyms

AAU Aalborg Universitet

AMBER Assessment of Metagenome BinnERs

ANN Artificial Neural Network

BP base pairs

CAE Convolutional Autoencoder

CAMI Critical Assessment of Metagenome Interpretation CAMI

CNN Convolutional Neural Network

DBSCAN Density-based spatial clustering of applications with noise

DCEC Deep Convolutional Embedded Clustering

DCMB Deep Convolutional Metagenomic Binner

DEPICT Deep Embedded Regularized Clustering

D_{KL} Kullback-Leibler divergence

DNA deoxyribonucleic acid

DSMB Deep Stacked Metagenomic Binner

DVMB Deep Variational Metagenomic Binnner

IMSAT Information Maximizing Self-Augmented Training

JULE Joint Unsupervised Learning

LPA label propagation algorithm

ORF open reading frame

ReLU Rectified Linear Unit

RIM Regularized Information Maximization

RNN Recurrent Neural Network

SAT Self-Augmented Training

t-SNE t-distributed stochastic neighbor embedding

VAE Variational Autoencoder

Bibliography

[1] Rosa Lundbye Allesøe Jakob Nybo Nissen Joachim Johansen and others. "Improved metagenome binning and assembly using deep variational autoencoders". In: *Nature* (2021). DOI: 10.1038/s41587-020-00777-4. URL: https://www.nature.com/articles/s41587-020-00777-4.

[2] Dongwan D. Kang et al. "MetaBAT 2: an adaptive binning algorithm for robust and efficient genome reconstruction from metagenome assemblies". In: (2019). DOI: https://doi.org/10.7717/peerj.7359.

[3] Ziye Wang et al. "SolidBin: improving metagenome binning with semi-supervised normalized cut". In: *Bioinformatics* 35.21 (Apr. 2019), pp. 4229–4238. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz253. eprint: https://academic.oup.com/bioinformatics/article-pdf/35/21/4229/30330800/btz253.pdf. URL: https://doi.org/10.1093/bioinformatics/btz253.

[4] CAMI. *1st CAMI Challenge Dataset 1 CAMI$_l$ow*. https://data.cami-challenge.org/participate. (Visited on 12/05/2021).

[5] Laura W Dijkhuizen. *Metagenomicspractical*. https://github.com/lauralwd/metagenomicspractical.

[6] Bojian Yin et al. "An Image Representation Based Convolutional Network for DNA Classification". In: *The International Conference on Learning Representations (ICLR)*. 2018.

[7] Henk Bolhuis et al. Laura W. Dijkhuizen Paul Brouwer. "Is there foul play in the leaf pocket? The metagenome of floating fern Azolla reveals endophytes that do not fix N2 but may denitrify". In: *The New Phytologist* 1 (2018), pp. 453–466. DOI: 10.1111/nph.14843.

[8] Kripa Adhikari, Sudip Bhandari, and Subash Acharya. "Reviews In Food And Agriculture (RFNA) AN OVERVIEW OF AZOLLA IN RICE PRODUCTION: A REVIEW". In: *Reviews in Food and Agriculture* 2 (Jan. 2021). DOI: 10.26480/rfna.01.2021.04.08.

[9] National Human Genome Research Institute. *Double Helix*. https://www.genome.gov/genetics-glossary/Double-Helix. (Visited on 04/08/2022).

[10] Cade Metz. *2016: The Year That Deep Learning Took Over the Internet*. https://www.wired.com/2016/12/2016-year-deep-learning-took-internet/. (Visited on 04/09/2022).

[11] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. DOI: 10.48550/ARXIV.1609.08144. URL: https://arxiv.org/abs/1609.08144.

[12] Chung-Cheng Chiu et al. "State-of-the-art Speech Recognition With Sequence-to-Sequence Models". In: 2018. URL: https://arxiv.org/pdf/1712.01769.pdf.

[13] El Allali A. Al-Ajlan A. "CNN-MGP: Convolutional Neural Networks for Metagenomics Gene Prediction". In: *Interdisciplinary Sciences, Computational Life Sciences* 11 (2018), pp. 628–635. DOI: 10.1007/s12539-018-0313-4.

[14] Antonino Fiannaca et al. "Deep learning models for bacteria taxonomic classification of metagenomic data". In: *BMC Bioinformatics* 19.7 (2018), p. 198. ISSN: 1471-2105. DOI: 10.1186/s12859-018-2182-6. URL: https://doi.org/10.1186/s12859-018-2182-6.

[15] Filip Wolf, Trong Dai Ha, and Jan Niklas Fichte. *Metagenomic Binning Based on Deep Learning*. 2021.

[16] Loris Nanni, Stefano Ghidoni, and Sheryl Brahnam. "Handcrafted vs. non-handcrafted features for computer vision classification". In: *Pattern Recognition* 71 (2017), pp. 158–172. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.05.025. URL: https://www.sciencedirect.com/science/article/pii/S0031320317302224.

[17] Andre Lamurias et al. "Metagenomic binning with assembly graph embeddings". In: *bioRxiv* (2022). DOI: 10.1101/2022.02.25.481923. eprint: https://www.biorxiv.org/content/early/2022/02/27/2022.02.25.481923.full.pdf. URL: https://www.biorxiv.org/content/early/2022/02/27/2022.02.25.481923.

[18] Tapinos A. Robertson Kouchaki S. "D.L. A signal processing method for alignment-free metagenomic binning: multi-resolution genomic binary patterns". In: *Scientific Reports* 9 (2019). DOI: 10.1038/s41598-018-38197-9.

[19] In: ().

[20] Yi Yue et al. "Evaluating metagenomics tools for genome binning with real metagenomic datasets and CAMI datasets". In: *BMC Bioinformatics* 21.1 (2020), p. 334. ISSN: 1471-2105. DOI: 10.1186/s12859-020-03667-3. URL: https://doi.org/10.1186/s12859-020-03667-3.

[21] DeepAI. *Hidden Representation*. 2019. URL: https://deepai.org/machine-learning-glossary-and-terms/hidden-representation.

[22] Rakesh Chada. *Understanding Neural Networks by Embedding Hidden Representations*. 2018. URL: https://medium.com/@rakesh.chada/understanding-neural-networks-by-embedding-hidden-representations-f256842ebf3a.

[23] Kamran Ghasedi Dizaji, Amirhossein Herandi, and Heng Huang. "Deep Clustering via Joint Convolutional Autoencoder Embedding and Relative Entropy Minimization". In: *CoRR* abs/1704.06327 (2017). arXiv: 1704.06327. URL: http://arxiv.org/abs/1704.06327.

[24] National Human Genome Research Institute. *DNA*. https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid. (Visited on 10/07/2021).

[25] Nature. *DNAreplication*. https://www.nature.com/scitable/topicpage/cells-can-replicate-their-dna-precisely-6524830/. (Visited on 10/07/2021).

[26] National Human Genome Research Institute. *chromosome*. https://www.genome.gov/about-genomics/fact-sheets/Chromosomes-Fact-Sheet. (Visited on 10/07/2021).

[27] Nature. *genome*. https://www.nature.com/scitable/definition/genome-43/. (Visited on 10/07/2021).

[28] National Research Council (US) Committee on Metagenomics: Challenges and Functional Applications. *The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet*. https://pubmed.ncbi.nlm.nih.gov/21678629/. The National Academies Press, 2007.

[29] National Human Genome Research Institute. *Genetic Marker*. https://www.genome.gov/genetics-glossary/Genetic-Marker. (Visited on 06/02/2022).

[30] Zhen Li et al. "Single-Copy Genes as Molecular Markers for Phylogenomic Studies in Seed Plants". In: *Genome Biol Evol.* 9.5 (2017), pp. 1130–1147. DOI: https://doi.org/10.1093/gbe/evx070.

[31] National Human Genome Research Institute. *DNA Sequencing*. https://www.genome.gov/genetics-glossary/DNA-Sequencing. (Visited on 09/30/2021).

[32] Miten Jain et al. "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community". In: *Genome Biology* 17.1 (2016). DOI: https://doi.org/10.1186/s13059-016-1103-0.

[33] Gunavaran Brihadiswaran. *Bioinformatics 1: K-mer Counting*. https://medium.com/swlh/bioinformatics-1-k-mer-counting-8c1283a07e29, urldate = 2021-12-28. 2020.

[34] Mehdi Kchou, Jean-François Gibrat, and Mourad Elloumi. "Generations of Sequencing Technologies: From First to Next Generation". In: *Biology and Medicine* 09 (2017), pp. 1–8. DOI: https://doi.org/10.4172/0974-8369.1000395.

[35] Yun Heo. "Improving quality of high-throughput sequencing reads". PhD thesis. University of Illinois, 2015.

[36] Jeff Illumina. *Read length recommendations*. https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/read-length.html.

[37] Nanoporetech. *MinION*. https://nanoporetech.com/products/minion. (Visited on 10/02/2021).

[38] David R. Greig et al. "Comparison of single-nucleotide variants identified by Illumina and Oxford Nanopore technologies in the context of a potential outbreak of Shiga toxin–producing Escherichia coli". In: *Giga Science* 8.8 (2019), pp. 1–12. DOI: http://dx.doi.org/10.1093/gigascience/giz104.

[39] Jason M. Neal-McKinney et al. "Comparison of MiSeq, MinION, and hybrid genome sequencing for analysis of Campylobacter jejuni". In: *Scientific Reports* 11.1 (2021), pp. 2045–2322. DOI: https://doi.org/10.1038/s41598-021-84956-6.

[40] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM*. 2013. arXiv: 1303.3997 [q-bio.GN].

[41] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (May 2018), pp. 3094–3100. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty191. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/18/3094/25731859/bty191.pdf. URL: https://doi.org/10.1093/bioinformatics/bty191.

[42] Christophe Oguey, Nicolas Foloppe, and Brigitte Hartmann. "Understanding the Sequence-Dependence of DNA Groove Dimensions: Implications for DNA Interactions". In: *PloS one* 5 (Dec. 2010), e15931. DOI: 10.1371/journal.pone.0015931.

[43] Rosa Lundbye Allesøe Jakob Nybo Nissen Joachim Johansen and others. "Supplementary information: Improved metagenome binning and assembly using deep variational autoencoders". In: *Nature* (2021). DOI: 10.1038/s41587-020-00777-4. URL: https://static-content.springer.com/esm/art\%3A10.1038\%2Fs41587-020-00777-4/MediaObjects/41587_2020_777_MOESM1_ESM.pdf.

[44] Shanrong Zhao, Zhan Ye, and Robert Stanton. "Misuse of RPKM or TPM normalization when comparing across samples and sequencing protocols". In: *RNA* 26 (Apr. 2020), rna.074922.120. DOI: 10.1261/rna.074922.120.

[45] Tom Taulli. *Artificial Intelligence Basics*. https://pubmed.ncbi.nlm.nih.gov/21678629/. Apress Berkeley, CA. DOI: 10.1007/978-1-4842-5028-0.

[46] Wissal Farsal, Samir Anter, and Mohammed Ramdani. "Deep Learning: An Overview". In: *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*. SITA'18. Rabat, Morocco: Association for Computing Machinery, 2018. ISBN: 9781450364621. DOI: 10.1145/3289402.3289538. URL: https://doi-org.zorac.aub.aau.dk/10.1145/3289402.3289538.

[47] Ameet Joshi. "Machine Learning and Artificial Intelligence". In: (Jan. 2020). DOI: 10.1007/978-3-030-26622-6.

[48] Martin Thoma. *File:Perceptron-unit.svg*. https://commons.wikimedia.org/wiki/File:Perceptron-unit.svg. (Visited on 03/20/2022).

[49] Riccardo Di Sipio. *A Quick Guide to Cross-Entropy Loss Function*. https://towardsdatascience.com/a-quick-guide-to-cross-entropy-loss-function-8f3410ec6ab1. (Visited on 06/16/2022).

[50] Jason Brownlee. *Softmax Activation Function with Python*. 2020. URL: https://machinelearningmastery.com/softmax-activation-function-with-python/.

[51] Harry McGrath et al. "Future of Artificial Intelligence in Anesthetics and Pain Management". In: *Journal of Biosciences and Medicines* 07 (Jan. 2019), pp. 111–118. DOI: 10.4236/jbm.2019.711010.

[52] Kiprono Elijah Koech. *Cross-Entropy Loss Function*. 2021. URL: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e#:~:text=Cross\%2Dentropy\%20loss\%20is\%20used.

[53] Bhadvika Kanani. *Cosine Similarity - Text Similarity Metric*. 2019. URL: https://studymachinelearning.com/cosine-similarity-text-similarity-metric/#:~:text=Cosine\%20similarity\%20is\%20one\%20of.

[54] Y. V. R. Nagapawan, Kolla Bhanu Prakash, and G. R. Kanagachidambaresan. "Convolutional Neural Network". In: *Programming with TensorFlow: Solution for Edge Computing Applications*. Ed. by Kolla Bhanu Prakash and G. R. Kanagachidambaresan. Cham: Springer International Publishing, 2021, pp. 45–51. ISBN: 978-3-030-57077-4. DOI: 10.1007/978-3-030-57077-4_6. URL: https://doi.org/10.1007/978-3-030-57077-4_6.

[55] Cornell University. *CS1114 Section 6: Convolution*. https://www.cs.cornell.edu/courses/cs1114/2013sp/sections/S06_convolution.pdf. (Visited on 05/25/2022).

[56] Vedant Kumar. *Convolutional Neural Networks*. https://towardsdatascience.com/convolutional-neural-networks-f62dd896a856. (Visited on 03/30/2022).

[57] Yugesh Verma. *https://analyticsindiamag.com/guide-to-different-padding-methods-for-cnn-models/*. https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/. (Visited on 04/03/2022).

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

[59] Dor Bank, Noam Koenigstein, and Raja Giryes. "Autoencoders". In: *CoRR* abs/2003.05991 (2020). arXiv: 2003.05991. URL: https://arxiv.org/abs/2003.05991.

[60] Hosameldin Ahmed, M. Wong, and Asoke Nandi. "Intelligent condition monitoring method for bearing faults from highly compressed measurements using sparse over-complete features". In: *Mechanical Systems and Signal Processing* 99 (Jan. 2018), pp. 459–477. DOI: 10.1016/j.ymssp.2017.06.027.

[61] Ons Aouedi, Kandaraj Piamrat, and Dhruvjyoti Bagadthey. "A Semi-supervised Stacked Autoencoder Approach for Network Traffic Classification". In: *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. 2020, pp. 1–6. DOI: 10.1109/ICNP49622.2020.9259390.

[62] Dr. PKS Prakash and Achyutuni Sri Krishna Rao. *R Deep Learning Cookbook R Deep Learning Cookbook*. https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781787121089/4/ch04lvl1sec51/setting-up-stacked-autoencoders. (Visited on 07/06/2022).

[63] Roger. *Variational Autoencoder(VAE)*. https://medium.com/geekculture/variational-autoencoder-vae-9b8ce5475f68. Accessed: 2021–04-28. 2021.

[64] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *CoRR* abs/1312.6114 (2014).

[65] Inductiveload. *A selection of Normal Distribution Probability Density Functions (PDFs). Both the mean, μ, and variance, σ^2, are varied. The key is given on the graph.* https://commons.wikimedia.org/wiki/File:Normal_Distribution_PDF.svg. (Visited on 10/08/2021).

[66] Ray Xiao. *The (Local) Reparameterization Trick*. https://www.cs.toronto.edu/~duvenaud/courses/csc2541/slides/structured-encoders-decoders.pdf. Accessed: 2021–10-08. 2016.

[67] Xifeng Guo et al. "Deep Clustering with Convolutional Autoencoders". In: Oct. 2017, pp. 373–382. ISBN: 978-3-319-70095-3. DOI: 10.1007/978-3-319-70096-0_39.

[68] Qien Yu, Muthu Subash Kavitha, and Takio Kurita. "Detection of One Dimensional Anomalies Using a Vector-Based Convolutional Autoencoder". In: Feb. 2020, pp. 516–529. ISBN: 978-3-030-41298-2. DOI: 10.1007/978-3-030-41299-9_40.

[69] Marco Maggipinto et al. "A Convolutional Autoencoder Approach for Feature Extraction in Virtual Metrology". In: *Procedia Manufacturing* 17 (2018). 28th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2018), June 11-14, 2018, Columbus, OH, USAGlobal Integration of Intelligent Manufacturing and Smart Industry for Good of Humanity, pp. 126–133. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.10.023. URL: https://www.sciencedirect.com/science/article/pii/S2351978918311399.

[70] Arwa Alturki, Ouiem Bchir, and Mohamed Maher Ben Ismail. "Joint Deep Clustering: Classification and Review". In: *International Journal of Advanced Computer Science and Applications* 12 (Jan. 2021). DOI: 10.14569/IJACSA.2021.0121096.

[71] Gopi Nutakki et al. "An Introduction to Deep Clustering". In: Jan. 2019, pp. 73–89. ISBN: 978-3-319-97863-5. DOI: 10.1007/978-3-319-97864-2_4.

[72] Dibya Ghosh. *KL Divergence for Machine Learning*. 2018. URL: https://dibyaghosh.com/blog/probability/kldivergence.html.

[73] Jianwei Yang, Devi Parikh, and Dhruv Batra. "Joint Unsupervised Learning of Deep Representations and Image Clusters". In: June 2016, pp. 5147–5156. DOI: 10.1109/CVPR.2016.556.

[74] Junyuan Xie, Ross B. Girshick, and Ali Farhadi. "Unsupervised Deep Embedding for Clustering Analysis". In: *CoRR* abs/1511.06335 (2015). arXiv: 1511.06335. URL: http://arxiv.org/abs/1511.06335.

[75] Md Karim et al. "Convolutional Embedded Networks for Population Scale Clustering and Bio-Ancestry Inferencing". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* PP (May 2020), pp. 1–1. DOI: 10.1109/TCBB.2020.2994649.

[76] Apache. *Apache/Spark: Apache Spark - a unified analytics engine for large-scale data processing*. URL: https://github.com/apache/spark.

[77] Yanick Lukic et al. "Speaker identification and clustering using convolutional neural networks". In: *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2016, pp. 1–6. DOI: 10.1109/MLSP.2016.7738816.

[78] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[79] IBM Cloud Education. *What are Recurrent Neural Networks?* 2020. URL: https://www.ibm.com/cloud/learn/recurrent-neural-networks.

[80] Mohammad Arifur Rahman and Huzefa Rangwala. "IDMIL: an alignment-free Interpretable Deep Multiple Instance Learning (MIL) for predicting disease from whole-metagenomic data". eng. In: *Bioinformatics (Oxford, England)* 36.Suppl_1 (2020). 32657370[pmid], pp. i39–i47. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btaa477. URL: https://pubmed.ncbi.nlm.nih.gov/32657370.

[81] Divyanshu Mishra. *Transposed Convolution Demystified*. 2021. URL: https://towardsdatascience.com/transposed-convolution-demystified-84ca81b4baba.

[82] Shiva Verma. *Understanding 1D and 3D Convolution Neural Network | Keras*. 2019. URL: https://towardsdatascience.com/understanding-1d-and-3d-convolution-neural-network-keras-9d8f76e29610.

[83] Shruti Jadon. "A survey of loss functions for semantic segmentation". In: *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. 2020, pp. 1–7. DOI: 10.1109/CIBCB48159.2020.9277638.

[84] Xuhuyang Guo. "Clustering of NASDAQ Stocks Based on Elbow Method and K-Means". In: *Proceedings of the 4th International Conference on Economic Management and Green Development*. Ed. by Chunhui Yuan, Xiaolong Li, and John Kent. Singapore: Springer Singapore, 2021, pp. 80–87. ISBN: 978-981-16-5359-9.

[85] Yang Young Lu et al. "COCACOLA: binning metagenomic contigs using sequence COmposition, read CoverAge, CO-alignment and paired-end read LinkAge". In: *Bioinformatics* 33.6 (June 2016), pp. 791–798. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw290. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/6/791/25147902/btw290.pdf. URL: https://doi.org/10.1093/bioinformatics/btw290.

[86] Mina Rho, Haixu Tang, and Yuzhen Ye. "FragGeneScan: predicting genes in short and error-prone reads". In: *Nucleic Acids Research* 38.20 (Aug. 2010), e191–e191. ISSN: 0305-1048. DOI: 10.1093/nar/gkq747. eprint: https://academic.oup.com/nar/article-pdf/38/20/e191/16772703/gkq747.pdf. URL: https://doi.org/10.1093/nar/gkq747.

[87] Jaina Mistry et al. "Challenges in homology search: HMMER3 and convergent evolution of coiled-coil regions". In: *Nucleic Acids Research* 41.12 (Apr. 2013), e121–e121. ISSN: 0305-1048. DOI: 10.1093/nar/gkt263. eprint: https://academic.oup.com/nar/article-pdf/41/12/e121/25338495/gkt263.pdf. URL: https://doi.org/10.1093/nar/gkt263.

[88] Bojian Yin et al. "An image representation based convolutional network for DNA classification". In: *arXiv preprint arXiv:1806.04931* (2018).

[89] Pablo Millán Arias et al. "DeLUCS: Deep learning for unsupervised clustering of DNA sequences". In: *Plos one* 17.1 (2022), e0261531.

[90] Allen Chieng Hoon Choong and Nung Kion Lee. "Evaluation of convolutionary neural networks modeling of DNA sequences using ordinal versus one-hot encoding method". In: *2017 International Conference on Computer and Drone Applications (IConDA)*. IEEE. 2017, pp. 60–65.

[91] National Human Genome Research Institute. *Open Reading Frame*. https://www.genome.gov/genetics-glossary/Open-Reading-Frame. (Visited on 05/21/2022).

[92] Scilico. *IUPAC Codes*. 2022. URL: https://www.bioinformatics.org/sms/iupac.html.

[93] Don Cowan. *Masking*. 2020. URL: https://www.ml-science.com/masking.

[94] MICROBIOME Community of Special Interest. *Welcome to the MICROBIOME Community of Special Interest*. https://www.microbiome-cosi.org/. (Visited on 05/21/2022).

[95] Alexander Sczyrba et al. "Critical Assessment of Metagenome Interpretation—a benchmark of metagenomics software". In: *Nature Methods* 14.11 (2017), pp. 1063–1071. ISSN: 1548-7105. DOI: 10.1038/nmeth.4458. URL: https://doi.org/10.1038/nmeth.4458.

[96] Fernando Meyer et al. "AMBER: Assessment of Metagenome BinnERs". In: *GigaScience* 7 (June 2018). DOI: 10.1093/gigascience/giy069.

[97] Lawrence Hubert and Phipps Arabie. "Comparing partitions". In: *Journal of classification* 2.1 (1985), pp. 193–218.

[98] Scikit Learn. *sklearn.metrics.adjusted$_r$and$_s$core*. 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html#:~:text=Rand\%20index\%20adjusted\%20for\%20chance.

[99] Donovan H Parks et al. "CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes". en. In: *Genome Res.* 25.7 (July 2015), pp. 1043–1055.

[100] Zhongwu Xie, Weipeng Cao, and Zhong Ming. "A further study on biologically inspired feature enhancement in zero-shot learning". In: *International Journal of Machine Learning and Cybernetics* 12 (Jan. 2021). DOI: 10.1007/s13042-020-01170-y.

[101] Andre Lamurias et al. “Metagenomic binning with assembly graph embeddings”. In: *bioRxiv* (2022). DOI: 10.1101/2022.02.25.481923. eprint: https://www.biorxiv.org/content/early/2022/02/27/2022.02.25.481923.full.pdf. URL: https://www.biorxiv.org/content/early/2022/02/27/2022.02.25.481923.

[102] Fernando Meyer et al. “AMBER: Assessment of Metagenome BinnERs”. In: *GigaScience* 7.6 (June 2018). giy069. ISSN: 2047-217X. DOI: 10.1093/gigascience/giy069. eprint: https://academic.oup.com/gigascience/article-pdf/7/6/giy069/25099086/giy069_supplemental_files.pdf. URL: https://doi.org/10.1093/gigascience/giy069.

[103] DH Parks et al. “CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes”. In: *Genome Research* 25 (2015), 1043–1055. DOI: 10.1101/gr.186072.114..