Aalborg University
The Technical Faculty of IT and Design

Title:

AutoBinner: A Metagenomic Binner combining Feature Learning and HDBSCAN

Project period:

P10, Spring 2020

03. February 2020 - 11. June 2020

Project group:

Group MI108F20

Group members:

Hansen, Mathias Lundhede

Linnebjerg, Simon Nagel

Vive, Lasse Erritsø

Supervisor:

Hose, Katja

Nielsen, Thomas Dyhre

Number of pages: 65

Abstract:

This report documents the development of the metagenomic binner ***AutoBinner***. Autobinner combines feature learning and clustering by using a stacked undercomplete autoencoder and the clustering algorithm HDBSCAN. The autoencoder learns a feature embedding, given abundance and composition features before the embedding is clustered.

To provide the full picture of AutoBinner the report also provides an explanation of the workings of neural networks in the context of autoencoders and feature learning.

The performance of AutoBinner is evaluated with a comparison with state of the art binners, MetaBAT2 [Kang et al., 2019], CONCOCT [Alneberg et al., 2013] and VAMB [Nissen et al., 2018], on three different datasets CAMI Medium, CAMI High, and CAMI Airways. The results indicate that further refinement of AutoBinner is needed, yet we see potential of using autoencoders and HDBSCAN for metagenomic binning.

Preface

This project is a continuation of the pre-specialisation project we did in the previous semester, see Linnebjerg et al. [2019]. [1]

Visualisations of the embeddings produced by the autoencoder, projected to 3-d as well as binning results can be found at: http://mi108.northeurope.cloudapp.azure.com/ [2]

We would like to thank Simon Rasmussen, co-author of Nissen et al. [2018] for providing access to the CAMI airways dataset.

We would also like to thank Prof. Mads Albertsen and PhD stipend Thomas Yssing Michaelsen from Albertsen Lab at the Center for Microbial Communities at Aalborg University for their guidance in understanding the field of metagenomic binning.

Last but not least, we would like to thank our two supervisors Prof. Thomas Dyhre Nielsen and Prof. Katja Hose, for feedback and guidance throughout this and our previous project.

[1] Note that it requires an AAU login to see the report

[2] The link expires after the 24th of June 2020

Summary

The aim of this project is to create a metagenomic binner comparable to state of the art metagenomic binners.

It is common for various binners to extract features from the abundance and composition features. In this project, we will look into feature learning methods allowing the extraction of salient features to represent each contig by learning an embedding using autoencoders.

Our problem analysis therefore has a main focus on showing the fundamentals of autoencoders. As an autoencoder is essentially a specific type of neural network we provide the foundation for autoencoders by making detailed descriptions of neural networks and their deep learning process. We describe different types of autoencoders, how they work, and their strengths and weaknesses with regards to feature learning. The last part of the problem analysis will touch upon two clustering algorithms that can be of use to cluster the features learned by the autoencoder. We will not go in great detail regarding the clustering process, as the clustering related intricacies regarding distance measures, similarity measures, etc. have been covered in our previous report. For the same reason, the problem analysis will not go into great detail with the problem domain, as this is also covered in our previous report.

Trough this report we present the binner, ***AutoBinner***. AutoBinner uses a stacked sparse autoencoder for feature learning from the aforementioned common contig representations. The embedding of the contigs are clustered using the HDBSCAN algorithm. To make the autoencoder learn an embedding that improves the clustering we propose a simple weighting scheme for controlling the weighting of the abundance and composition features for the learning process of the autoencoder. Our experiments show that this weighting is important for proper feature learning as the features are not of equal importance when determining which contigs originated from the same genome.

To validate that AutoBinner is working as intended, we make an initial set of experiments on a medium complexity dataset with a simple version of the binner. Here we show that AutoBinner is capable of learning an embedding of the common contig representations that improves the number of high quality bins.

We tweak the parameters for both the autoencoder and the clustering algorithm to yield high quality results on the dataset CAMI high. Here we also document our process of combining a clustering objective and the reconstruction objective of the autoencoder to form a joint optimisation process that makes the autoencoder learn an embedding that optimises both objectives simultaneously. However, the joint optimisation did not improve the binning results.

We compare AutoBinner against other binners; MetaBAT2, CONCOCT, and VAMB. When comparing the binners on the dataset CAMI High used for parameter-tuning, we see that AutoBinner outperforms the other binners by a large margin. However, taking

the comparison to a more fair dataset, CAMI Airways, the results show that AutoBinner falls behind.

In the discussion part of the report, we take up some of the shortcomings of our project. We discuss different problems with AutoBinner that causes it to not perform well on the CAMI Airways dataset as well as possible improvements.

As our initial idea of incorporating a clustering loss into the reconstruction loss of an autoencoder was not successful, we show that AutoBinner can do reasonable binning. Despite getting mixed results on the various datasets, we see that small tweaks to the model yielded improvement on the validation datasets. Through our tests, we see that there definitely is potential for a binner using an autoencoder for learning features, prior to clustering with the HDBSCAN algorithm.

Table of contents

1 Introduction

When examining a biological environment, it is often of high interest to determine the genomes present in the environment. The fields in which these can be beneficial are many e.g. within agriculture it can be of interest to examine the soil of crops to enhance their living conditions while it within disease diagnosis is of interest to examine the structure of a deadly virus such as Covid-19 to find a cure. [National Research Council; Division on Earth and Life Studies, 2007]

A common first step on the way of determining genomes present in metagenomic samples is a process called shotgun sequencing. In this process, the complete genomes are split into multiple small sequences of DNA and must be put back together to be of use. These DNA sequences consist of nucleotides, i.e. the bases that form the structure of the genomes, indicated by the letters *A, T, C,* and *G*. The sequence of the four bases is determined by a sequencer and is then called a read. The reads are given to an assembler that puts the reads together. However, the assembler cannot reconstruct the entire genome but only larger sequences called contigs. These contigs are the main input to a binner. Binning a metagenomic sample is then the process of grouping together the contigs originating from the same genome. The goal is that the contigs in the same bin can be put together and in that way determining which genomes are present in the sample.

The binning process is illustrated in Figure 1.1. The final result of the binning process is called the metagenome, which is the complete set of DNA that describes all organisms in the sample.

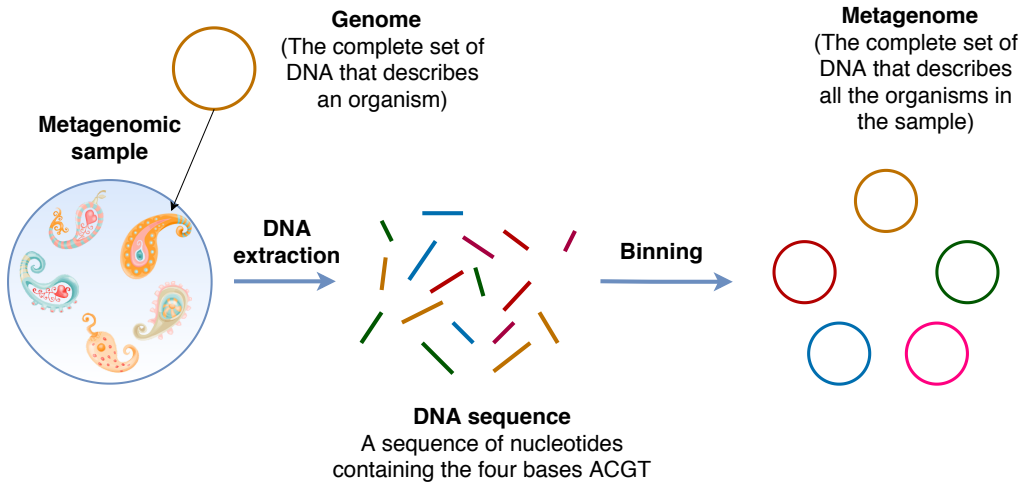

Figure 1.1: Illustration of the binning process

A metagenomic sample can give rise to hundreds of thousands of contigs consisting of several thousand or millions of nucleotides. A contig is commonly represented with two features: the composition features and abundance features.

Some binners do not cluster directly on the abundance and composition features, but rather on features extracted from the abundance and composition features. Multiple methods of extracting features suitable for binning have been proposed e.g. using PCA [Alneberg et al., 2013], or t-SNE [Lin and Liao, 2016]. A recent study [Nissen et al., 2018] has shown promising results by learning salient features from the abundance and composition features using a variational autoencoder trained to reconstruct the original two sets of features.

In this report we explore the feature learning capabilities of autoencoders in the domain of metagenomic binning, to learn features suited for clustering.

Thereby we introduce ***AutoBinner***, a binning tool that uses an autoencoder to create an embedding of the contigs, before clustering the embedded representations using the HDBSCAN algorithm [McInnes et al., 2017].

Problem analysis 2

In this chapter we will make a description of contig representations in section 2.1, which are commonly used in the field of metagenomic binning, and which will be used in this project. We will describe the benefits of feature learning in section 2.2. In regards to feature learning, we will provide explanations of neural networks and autoencoders in section 2.3 and respectively. In section 2.6 we will explain how clustering and feature learning can be combined to form "deep clustering" methods that uses a joint optimisation procedure to learn a feature embedding well suited for clustering.

2.1 Contig representation

In the context of metagenomic binning, it is common to represent each contig by the use of abundance and composition features. Abundance features describe the relative abundance of the contig in metagenomic samples, and the composition features describe the structure of the contig.

The composition features are extracted from each 4-mer of bases in the contig. As a contig is a sequence of four different bases, there is a total of $4^4 = 256$ unique 4-mers. Removing all reverse complement 4-mers, e.g. "GCAC" is the reverse complement of "CACG", leaves 136 4-mers (including palindromic 4-mers such as "GCCG" and "AAAA"). Each of the 4-mers in the contig is counted to give the composition feature illustrated in Table 2.1. Here each composition value is indexed with the corresponding 4-mer for intuition purposes, however, the feature vector consists only of the values such that each contig has a composition vector of length 136.

The composition features are l1-normalised across contigs to account for contigs being of different length, as a long contig will inherently have a higher count of each 4-mer than a short contig. The composition feature of a contig i.e. the distribution of the 136 4-mers of the contig can intuitively be thought of as being its language. Contigs with the same composition (language) will be more likely to originate from the same genome, and should thus more likely be binned together.

The abundance feature is created by using multiple metagenomic samples from the same environment. Having a set of reads for a number of samples and a set of contigs allows for measuring how prevelant each contig is in each sample i.e. how many reads from a given sample can be mapped to the contig. A simplified version of the mapping process is shown in Figure 2.1. The contig in the top of the figure is having its abundance measured in the sample from which the three reads originate. The abundance level of the contig is the sum of abundance per base divided with the length of the contig i.e. $\frac{1+2+3+3+2+1}{6} = 2$ This process is repeated for all available samples and thus creates a feature vector with

abundance values for each contig with length equal to the number of samples. When the number of mapped reads for each contig is counted, the abundance vector is l1-normalised. The normalisation makes the feature indicate the change in abundance across each sample.

4-mer	Count
"AAAA"	135
"AAAC"	236
...	...
"GCAG"	140
"GCAT"	340
...	...

Table 2.1: The composition feature vector containing counts of each 4-mer before normalisation.

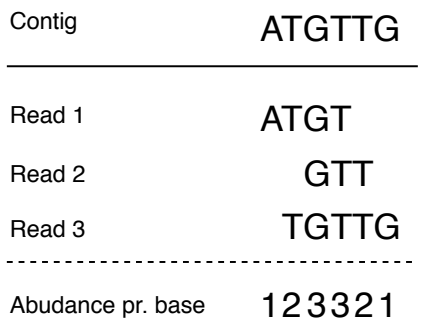

Figure 2.1: An example where three reads from the same sample of varying length are mapped to a contig. The abundance value for the contig in the sample is the average of the abundance pr. base. Figure is from Linnebjerg et al. [2019]

The intuition of the abundance feature is, that contigs with correlated abundance values should be binned together. The stronger the correlation, the more likely it is that they belong to the same genome. A fictional example of how abundance values can vary across samples is shown in Figure 2.2. Here it is clear that the green and yellow contigs are correlated across samples and that green and blue are not. Thus green and yellow should be binned together as well as blue and red.

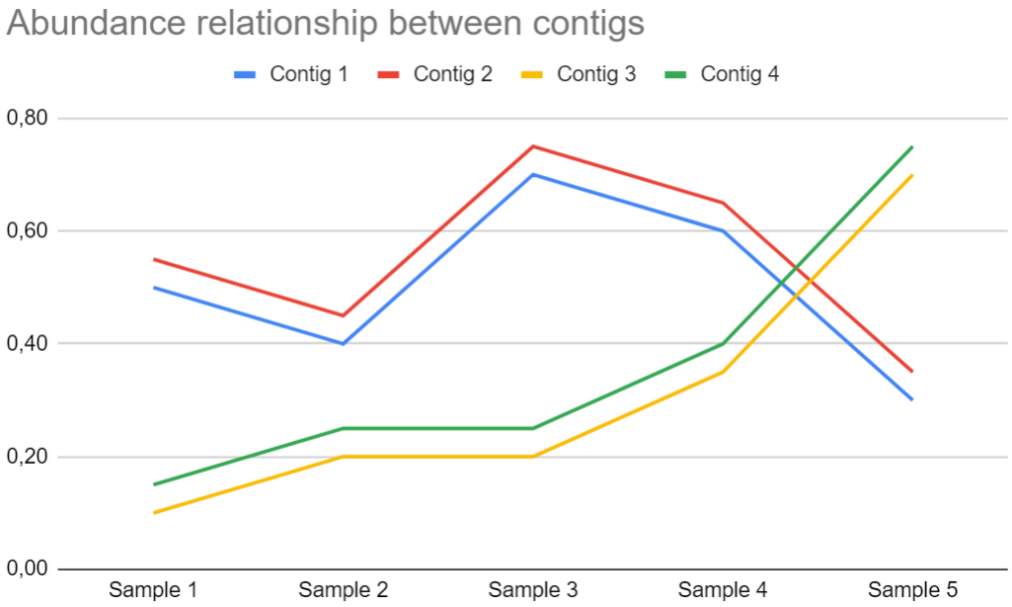

Figure 2.2: Plot of the abundance values of four contigs for five samples. The correlated contigs across samples should be binned together.

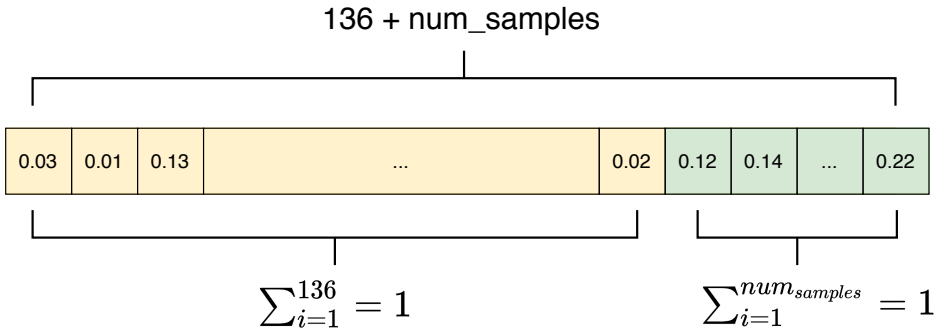

Figure 2.3: Illustration of the contig representation. Here the yellow fields represents the composition and the green fields represent the abundance.

The composition and abundance features are combined for each contig to form a feature matrix of size $N \times (136 + Num_{samples})$, where N is the total number of contigs, 136 is the number of 4-mers, and $Num_{samples}$ is the number of samples in the dataset. An example of the feature vector for a single contig is shown in Figure 2.3.

2.2 Feature learning

Having high-dimensional representations for each contig, i.e. $136 + Num_{samples}$, can have multiple drawbacks. First of all, high-dimensional data requires more space and is computationally more expensive to use than its counterpart. But more important some of the dimensions might not contribute to discriminating between the contigs.

A widely used approach for feature learning is principal component analysis (PCA). The main idea of PCA is to find a linear transformation, mapping the data to a lower dimensional space while preserving as much variance in the data as possible to not lose excessive information. PCA has its foundation in linear algebra and eigendecomposition and thus the mapping of the high dimensional data to a low dimensional embedding is linear. Therefore PCA is not able to find non-linear correlations between the dimensions which restrict the expressiveness of the embedding.

Another approach allowing for a non-linear embedding from one space to another, is to use an autoencoder. An autoencoder consists of two parts; an encoder and a decoder. The responsibility of the encoder is to learn a mapping, i.e. a function, that given some input gives a lower dimensional embedding. In contrast, the decoder must learn a mapping that, given the embedding, can reconstruct the original input. A conceptual illustration of an autoencoder can be seen in Figure 2.4, where the arrows f_θ and g_θ correspond to the encoder and decoder respectively. The illustration should be read as; given some input $\mathbf{x}$, the encoder f, parametrized by θ, outputs an embedding $\mathbf{h}$. The embedding is then fed to the decoder g, parametrized by θ, which gives the reconstructed input $\mathbf{y}$. The autoencoder is evaluated by how well it can reconstruct the input, e.g. by the sum of squared errors, and therefore it is a learning process to find the parameters θ that create a good embedding, which can be reconstructed, and in the case of binning, also a good clustering candidate. The metagenomic binner VAMB [Nissen et al., 2018] uses a variational autoencoder to extract features from the contig representations, and has shown

state of the art results. The encoder and decoder can be combined into a single function representing an autoencoder. An autoencoder is commonly modeled as a special type of function called a neural network which allows for non-linearity and which objective is not determined by preserving variance. This makes the autoencoder more expressive than PCA and a good candidate for feature learning.

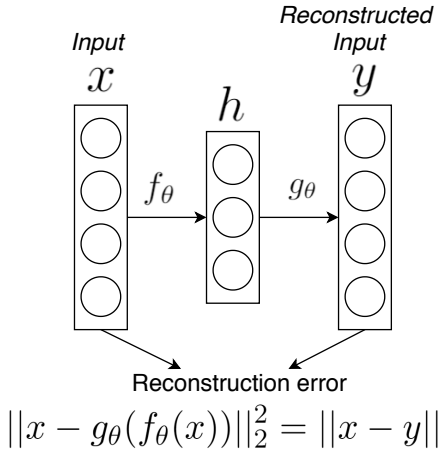

Figure 2.4: A simple autoencoder

In order to get a better understanding of the inner workings of autoencoders, we will in the following section provide an in-depth description of neural networks (NN) and how they learn their parameters.

2.3 Neural networks

In this section, we describe the basics of neural networks and specifically feedforward neural networks with the aim of connecting them to autoencoders. For the remaining part of this section, it is inferred that we discuss feedforward neural networks. In this section, we will cover the basics of a neural network, how data is propagated through the network to form an output. This covers weights and biases in section 2.3.1 and activation functions in section 2.3.2. When the overall structure of a NN is covered, we move on to how a NN learns in section 2.3.3. Specifically, the subjects discussed are loss functions, gradient descent, stochastic gradient descent, backpropagation, optimizers, regularisation and weight initialisation.

2.3.1 Structure of a neural network

A simple interpretation of a neural network is that it is a function that given some input produces an output. The output of the neural network depends on its usage which among others includes prediction tasks and feature learning in the form of an autoencoder. More formally we can write a neural network as $f_\theta(x) = y$ where the function f has the parameters θ and takes the input x to produce output y. An illustration of a simple neural network can be seen in Figure 2.5. [Géron, 2019]

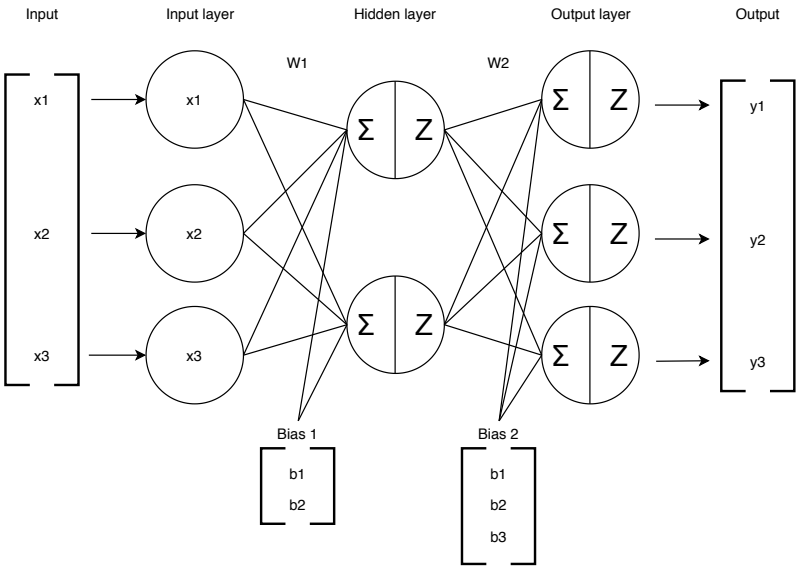

Figure 2.5: A simple feedforward neural network. The network can be seen as an autoencoder, where the left side is the encoder, the right side is the decoder and the hidden layer in the middle is the embedding.

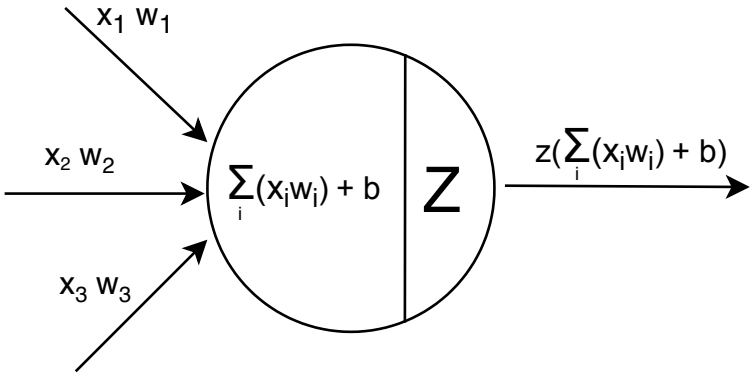

Figure 2.6: Illustration of the computation at a neuron. The x's are the output of the previous layer and the w's are the ingoing weights. The product of the x's and w's are summed and the bias of the neuron is added. Finally, the activation function z is applied giving the output of the neuron.

The network has three layers: an input layer, a hidden layer, and an output layer. Each layer consists of a number of nodes called neurons. In Figure 2.5 the input layer has three neurons, as the input, in this case, is a vector of three elements. The hidden layer has two neurons while the output layer has three neurons, corresponding to the number of inputs. Between the nodes of each layer, there is a number of weights represented as lines in the figure. The layers in the figure are called fully connected layers as there exists a weight between each neuron in the previous layer to each neuron in the next layer. In addition to the weights, each neuron has a bias value, except for the neurons in the input layer. In the figure, the biases are represented as a vector of values for each layer, where each value is associated with a neuron.

As shown in Figure 2.6, the input to the neuron is the output values of all neurons in the previous layer $x_1, ..., x_n$, and a set of weights $w_1, ..., w_n$, one for each incoming value. Within each neuron two operations take place. First, the incoming values are multiplied with their respective weight and summed up before a bias value b is added to the sum. Afterward, an activation function is applied to the sum, symbolized by z in the figure. The calculations performed in a neuron can thus be summarised with Equation 2.1 where I is the number of incoming values from previous the layer.

$$z((\sum_{i=1}^{I} x_i w_i) + b) \tag{2.1}$$

When data is fed to the neural network it passes from left to right such that the output from one layer is the input of the next layer. We can formalise the computation in each layer by using linear algebra giving the equation in Equation 2.2, where the values for a layer h are calculated. The weights going into a given layer can be represented as a matrix W that has a row for each input neuron and a column for each neuron in the in-going layer. E.g. the weight matrix W_1 in the figure would be of the dimensions 3×2. The bias vector for a given layer is represented as b having a value for each neuron in the layer.

$$h_{W,b}(x) = z(\mathbf{x}W + b) \tag{2.2}$$

In the hidden layer, the $\mathbf{x}$ corresponds to the neurons of the input layer, while it for the output layer corresponds to the output of the hidden layer. In the equation it can be seen, that the activation function is applied after the matrix multiplication, which is a result of the practice, that the same activation function commonly used for all the neurons in a layer. By chaining these computations for each layer the input is propagated forward till an output is produced.

As the hidden layer has fewer neurons than the input layer it allows us to use the neural network as an autoencoder, where the output of the hidden layer is the embedding. We can see the first transformation, i.e. when data is propagated from the input layer to the hidden layer as the encoder while the second transformation, i.e. transforming the data from the hidden layer to the output layer as the decoder.

2.3.2 Activation functions

It can be seen in Equation 2.2 that the computations in the forward pass are simply a combination of linear transformations, one for each layer, but with one addition - an activation function. The activation function is an important addition because it can introduce non-linearity into the neural network and thereby increase its expressiveness. Two commonly used activation functions are the Sigmoid and ReLU activation functions shown in Figure 2.7 and Figure 2.8.

The Sigmoid activation function, given by Equation 2.3, returns a value between 0 and 1. The smaller the input value the closer it will be towards 0, while the larger the input the closer the output will be to 1. This property makes the output of neurons intuitive to understand as values near zero means that very little signal is propagated forward to the next layer, while the opposite also holds.

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

The ReLU activation function, given by Equation 2.4, has the range $(0, +\infty)$ where a negative input will be mapped to 0 and a positive input simply will be the output as well. This means that contrary to the Sigmoid activation function which always allows some signal to pass through, ReLU cuts off all outgoing signal when the input is below the threshold. Due to this approach, ReLU is computationally simpler than the Sigmoid activation. In addition, it has been shown to make the training process of the neural network faster. [Li et al., 2020a]

$$ReLU(x) = max(0, x) \tag{2.4}$$

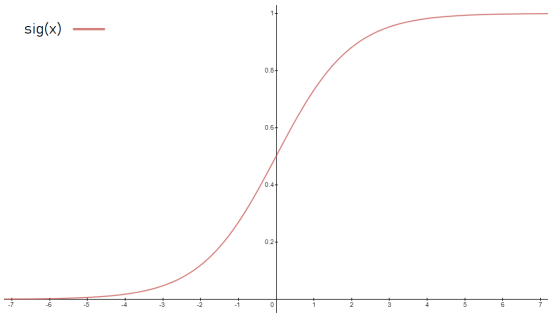

Figure 2.7: Graph of the Sigmoid activation function.

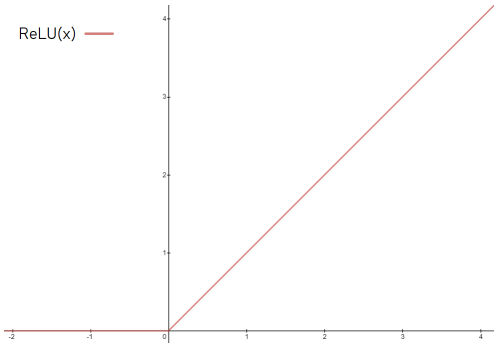

Figure 2.8: Graph of the ReLU activation function.

Looking at the activation functions can also help to understand the importance of having a bias term. In Figure 2.10 we can see that the weight of w has an effect corresponding to increasing or decreasing the steepness of the function, where the larger the weight the steeper the curve. However, common for the three functions in Figure 2.10 is that all inputs in the range of $-\infty, 0$ will equal 0, i.e. their threshold is 0. This restricts the network as it might be the case that the threshold instead should have been of some other value. The solution to this problem is to include a bias term that enables the functions to be shifted. An example can be seen in Figure 2.9 where the weight is one and three different bias terms are added. Hereby the steepness is preserved while the threshold is changed.

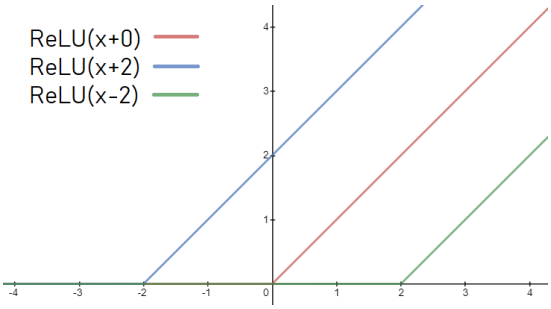

Figure 2.9: The effect of the bias term in regards to RelU

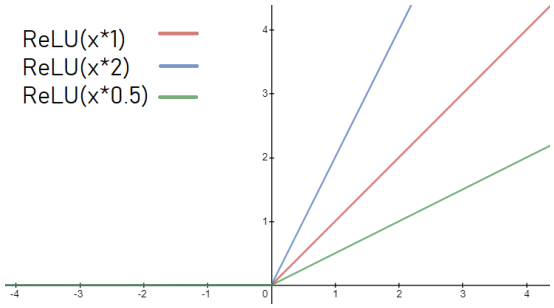

Figure 2.10: The effect of the weight term in regards to RelU

2.3.3 Learning

As we have provided an understanding of how the neural network is structured with regards to forward propagation, the weights, biases, and activation functions, we will take a look at how the neural network learns to produce meaningful output by training. The objective of training a neural network is ultimately to reduce the loss of the model with respect to the training data. To achieve this goal the weights and biases of the network are trained to fit the input data better and produce better reconstructions, predictions, etc. So what one wants to do is to find a direction in the weight space, i.e. changes to the weights, that yields a lower loss. The direction to change the weight vector will thus be related to the gradient of the loss function.

The following sections cover the different aspect of the NN learning process. In section 2.3.3.1 loss functions relevant for autoencoders are discussed. Section 2.3.3.2 describes the gradient descent algorithm to update the weights of the neural network. Section 2.3.3.3 describes how the specific weight updates are derived through backpropagation. Section 2.3.3.4 discusses why and how the stochastic gradient descent algorithm is a more efficient alternative to gradient descent. We then discuss some problems and solutions with weight initialisation in section 2.3.3.7. Lastly, we will discuss a commonly occurring problem with neural networks; overfitting, as well as solutions to overfitting, called regularisation, in section 2.3.3.8, and 2.3.3.9 respectively.

2.3.3.1 Loss function

The loss function is used to evaluate the performance of a neural network, and inform it how wrong it is so it can improve. When training a neural network the goal is to minimise the loss of the network. This means that the objective of the loss function is to whittle down all the aspects of the network, down to a single number, that can then be used for optimising the network. That number is then also an indicator that tells if the network is improving or not.

Suppose we have a neural network that should output its input as close as possible, and that the middle layer has fewer neurons than the input has dimensions (i.e. an autoencoder).

The goal of an autoencoder is to find an embedding of some input vector X which can be reconstructed as the vector Y from the embedding, such that the differences between X and Y are minimum. An intuitive loss function is the Mean Absolute Error (MAE) shown

in Equation 2.5, which calculates the absolute differences between X and Y where D is the number of dimensions in input/output vectors and N is the number of samples in the dataset.

$$MAE(X,Y) = \frac{1}{N}\sum^{N}\sum_{d=1}^{D}|X_d - Y_d| \tag{2.5}$$

Another commonly used loss function for the problem of reconstruction is the mean squared error (MSE) shown in Equation 2.6.

$$MSE(X,Y) = \frac{1}{N}\sum^{N}\sum_{d=1}^{D}(X_d - Y_d)^2 \tag{2.6}$$

The main difference between the two losses is the squaring of the errors. Squaring the error in MSE has the effect of giving points that are hard to handle for the network, a large effect on the loss. Therefore one factor in deciding whether to use MSE of MAE depends on how outliers should be treated.

Choosing a suitable loss function for the problem at hand is of high importance for the network to learn properly. However, a clear cut definition of when each is useful is not possible. Choosing the right loss function is a matter of understanding the problem domain, and its connection to the architecture of the neural network [Aggarwal, 2018,sec. 1.2.1.5].

2.3.3.2 Gradient descent

The optimisation problem lies in how the two parameters should be altered to minimise the loss. Starting from some point on a loss surface, dictated by the initialisation of the parameters, the idea is to calculate the direction the parameters should be altered to cause the steepest decrease in the loss. This method is called gradient descent and is commonly used in optimisation problems and neural networks. Li et al. [2020e]

Once the gradient of the loss function with respect to the parameters is calculated one simply nudges all weights in the opposite direction of the gradient, as the gradient shows the direction of the largest increase of the loss. Gradient descent is thus a fairly simple algorithm that updates the weights using Equation 2.7 where $\frac{\partial f}{\partial W_t}$ is the gradient of the loss function f with respect to the weights W in timestep t and α is a hyperparameter controlling how large a step to take for the weights in the direction of the negative gradient. Li et al. [2020e]

$$W_{t+1} = W_t + \alpha * (-\frac{\partial f}{\partial W_t}) \tag{2.7}$$

2.3.3.3 Backpropagation

In backpropagation, the chain rule is applied recursively to compute gradients of expressions. Here we have that f is again the loss function evaluating the parameters by applying the parameters to the input and forming a loss value. The parameters are both the weights W and biases $\mathbf{b}$.

The partial derivatives are calculated in a simple manner requiring only simple calculus. Complicated compound expressions such as $f(x, y, z, u) = (x * y)(z + u)$ are easily handled by splitting the expression up in the most simple components, and a computation graph can be helpful to keep track of the computation more easily. The computation graph expressing the compound expression f is shown in Figure 2.11. In the figure, the expression is broken down into three expressions $q = x + y$, $p = zu$, and $f = qp$, however, we are only interested in the gradient of f with respect to its inputs x, y, z and u. The variables have the values: $x = 2, y = 3, z = 4, u = 2$, the green numbers on the edges are thus the variable values and the results of expressions q, p, and f. The red numbers are the partial derivatives that we are interested in finding. Li et al. [2020b]

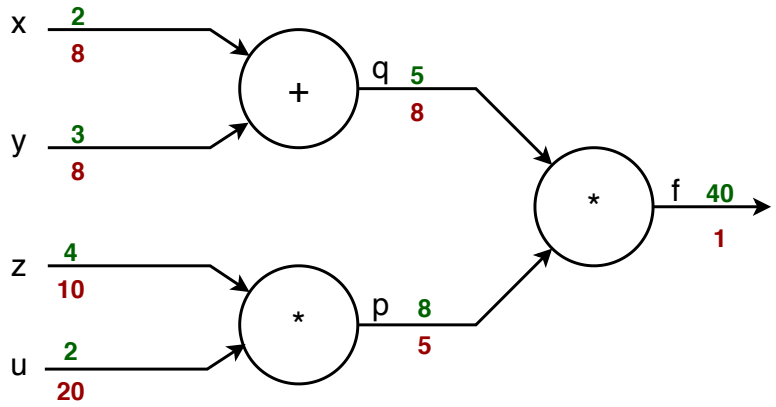

Figure 2.11: The computation graph for expression $f(x, y, z, u) = (x + y)(z * u)$.

When multiple expressions are combined, the correct way to chain the gradient expressions together is by using the chain rule. The gradient for function f with respect to variable x in the example is thus calculated as:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Starting from the right side with the output of f, the gradient $\frac{\partial f}{\partial f}$ is simply 1, the trivial case. This "local" gradient in the node f is then flowing "up-stream" through the graph and is used for calculating the other gradients. The gradient of p can then be calculated with the chain rule. The gradient of p is 5, as q has a down stream value of 5, which is multiplied with 1, the gradient of f:

$$\frac{\partial f}{\partial p}\frac{\partial f}{\partial f} = 5 * 1 = 5$$

Likewise, the gradient of variable z is calculated using its local gradient, which is equal to the value $u = 2$, and the upstream gradient from p.

$$\frac{\partial f}{\partial z} = \frac{\partial p}{\partial z}\frac{\partial f}{\partial p} = 2 * 5 = 10$$

Intuitively, it can be seen why this makes sense. Increasing z with 1 has an effect with magnitude $u = 4$ (multiplication) on the output of p, which in turn again affects the output of f, this time with magnitude 5, such that the effect of increasing z has the effect: $(1 * u) * 5 = (1 * 4) * 5 = 20$. Li et al. [2020b]

And thus, the partial derivative of each parameter in a neural network can be calculated quite easily as a neural network can be represented as a complex computation graph. The computation graph of a neural network would have all the weights, biases, and the specific input being fed to the neural network as its input. The values would be passed down-stream through all the intermediate nodes till the rightmost computation node corresponding to the loss function would be reached. Afterwards, the gradients would be calculated from right to left such that intermediate gradients can be reused. This procedure is repeated till the gradients of the network parameters have been calculated. Let W be the all the parameters of the neural network with the loss function f. The parameters are then updated by Equation 2.8

$$W_{updated} = W + \alpha(-\Delta f_w) \tag{2.8}$$

Using this method of splitting complicated expressions into multiple smaller expressions makes it quite intuitive and easy to see how gradients are calculated for each input, weight, and bias. The method thus easily scales up to cover an entire NN with many large layers.

2.3.3.4 Stochastic gradient descent

When using gradient descent, the parameters of the neural network are only updated when gradients have been calculated for all samples in the dataset. A full pass of all data in the training set is called an epoch. The parameters are then updated by the average of all the gradients. This has the advantage that the update of the parameters is precise but it has the drawback of being computationally heavy. A common way to update the parameters more often and thereby having the neural network converge faster is to use mini-batch gradient descent. In mini-batch gradient descent the dataset is split into several batches with the intuition that if a batch is representative of the overall dataset, the network can be updated by the average gradient of a batch. Mini-batch gradient descent is commonly referred to as stochastic gradient descent (SGD). It is called "stochastic" as each batch is a randomly selected subset of the training data. The batches are created randomly at every epoch to ensure that the batches are composed differently at each epoch. This avoids that a certain batch contains too much identical data which could slightly throw off the weights every time said batch is encountered, and that the network does not learn the order of the batches. Li et al. [2020e]

2.3.3.5 Momentum

The classical example used to explain momentum is to imagine the loss function as a hill, and the current loss as a ball that should get to the bottom of the hill, see Figure 2.12. By computing the gradients of the loss function the direction to 'push' the ball down the hill is determined. However, without momentum, it is like the ball only rolls when it is being actively pushed, and when there is no one pushing (the direction of the gradient is not very steep), it comes to an immediate stop. Momentum looks at the previous gradient that was used to push the ball, and then adds it on to the current gradient to push the ball even harder down the hill. This has the advantage of helping avoiding the loss getting stuck in a local minimum, as the momentum will keep the ball rolling. If we then look at SGD with momentum added we get the following

$$W_{updated} = \mu * W_{prev} + \alpha(-\Delta f_w) \tag{2.9}$$

Here $\mathbf{W}_{prev}$ is the update from the previous weight update. Another variable added is μ which is a way to control the momentum, such that the ball stops when it reaches the bottom of the hill, i.e. the global minima, and does not just keep going up another hill. In some cases, it can be beneficial to increase μ in the later stages of the training, to ensure that the loss stops at the global minima.

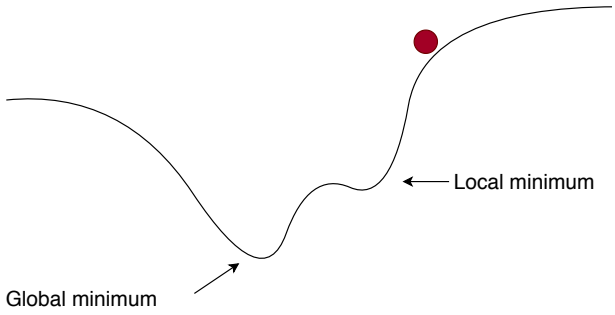

Figure 2.12: The classic example of momentum, using a ball that is rolling down a hill.

2.3.3.6 Adam

Adam is a method for updating the parameters of a network, by looking at the exponential moving average of the gradient. The exponential moving average is chosen over the regular moving average in order to be able to weight the more recent gradients higher.

Compared to SGD, which only takes the learning rate α as input parameter, Adam takes four input parameters; the learning rate α, two biases $\beta 1$ and $\beta 2$, and ϵ. $\beta 1$ and $\beta 2$ controls the exponential decay rates of the moving averages of the gradient, and ϵ is a very small value that is introduced to avoid dividing by zero when updating the parameters, as we will see below. Adam works by computing adaptive learning rates for the different network parameters from estimates of the first and second moments of the gradient's moments, i.e. the exponential moving average of the gradient m and the squared gradient v. To do so, it repeats the following steps until the parameters converge.

1. Compute the gradients
2. Update the estimate of the biased first and second moment v and m

$$m = \beta 1 * m_{prev} + (1 - \beta 1) * \Delta W$$
$$v = \beta 2 * v_{prev} + (1 - \beta 2) * (\Delta W^2)$$

3. Compute the estimate of the bias-corrected first and second moment $\hat{v}$ and $\hat{m}$

$$\hat{m} = \frac{m}{1 - \beta 1}$$
$$\hat{v} = \frac{v}{1 - \beta 2}$$

4. Update the parameters

$$W+ = -\alpha * \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

In the second step, we see that the momentum comes into play, as in order to find the first and second moment v and m, the first and second moment v_{prev} and m_{prev} from the previous gradient update are used. In The bias correction that happens in step three is the vectors m and v initialised at zero, so to get them 'going' faster, and not be biased around zero, this step tries to remove that bias. In the last step, we see that ϵ is added to the square root of v, which is done in case that v is equal to zero, which would result in a division by zero. ϵ usually is very low value, where Kingma and Ba [2015] recommends that ϵ should be 10^{-8}.

2.3.3.7 Weight initialisation

The weight initialisation is the starting point of the neural network. Suppose that the training data is zero centered. Thus, it makes sense to assume that initialising all weights to zero will yield decent initial losses as the mean is a pretty good guess. However, if all weights are initialised to the same value, the network is not able to learn properly. If the weights are all the same, then the partial derivatives would be equal in many places, and the weights would be updated identically. The neurons are then said to be symmetric, and many neurons would be doing the same thing in different places. Li et al. [2020c]

It is better to keep the initial weights close to zero but with random values close to zero to breaks the symmetry e.g. the weights could then be sampled from a zero mean Gaussian distribution with unit standard deviation, $G(0, 1)$. When the symmetry is broken the weights are updated in distinct ways and can again be integrate with each other to form different meaningful parts of the network. However, it is not certain that it is better to initialize weights close to zero. When passing data through the network, where all the weights are very low, the activations (the values coming out of a neuron) will be smaller and smaller for each layer and collapse near zero. When the activations become very small and fed forward to a neuron then the gradients of said neuron will also be very small. Thereby the weights will only be updated by a very small amount and the network will not learn. Li et al. [2020c]

On the other hand, large weights are not necessarily preferred either. With large weights, the input to an activation function will always be very positive or very negative, and for some activation functions, such as Sigmoid, these two extremes are problematic due to the problem of vanishing gradients. When the input is too positive or too negative for Sigmoid or TanH, the gradients of the function will be close to zero, and again this is not good for learning as the updates to the weights will be too small and the network will not learn anything. Li et al. [2020c]

The weights must not be too small nor too large. What instead has shown to work well is using "Xavier initialisation". Xavier initialisation also samples the weights from a zero mean Gaussian distribution with unit standard deviation: $G(0, 1)$, but the values are scaled by dividing with the square root of the number of inputs to the neuron n.

$$w = sample(G(0, 1))/\sqrt{n}$$

This makes the weights behave such that the variance of the output values is the same as the variance of the input values. If a neuron has a small number of inputs, the weights need to be larger to achieve the same variance. If the neuron has a large number of inputs the weights need to be smaller to get the same variance at the output. Having the same variance for each input/output throughout the network ensures that the values computed when passing the values from input to output do not collapse to zero, and also helps alleviate the vanishing gradients problem. Li et al. [2020c]

However, Xavier initialisation does not solve all problems. When using the ReLU activation function, this initialisation is not good enough. ReLU essentially turns (approximately) half of the incoming activations off (the negative inputs), and thereby the output variance is essentially cut in half. This again makes the distributions of activations collapse towards zero. Xavier initialisation can be tweaked to adjust for ReLU shutting off half the input. The tweaked version scales only by $2.0/n$ and is called "He" (or "He-et-al") initialisation:

$$w = sample(G(0, 1))/\sqrt{2/n}$$

The biases on the other hand are commonly initialised to zero. This can be allowed as the symmetry breaking is already ensured by the weights.

As mentioned in subsection 2.3.2 ReLU has become a preferred activation function, however, it can face a problem during training called the 'dying RelU' problem. Too large gradient updates during backpropagation can cause weight updates that are too drastic such that inputs never are to get activated, i.e. the output of the neuron will always be zero and thereby the neurons provide nothing useful for the network and are essentially dead. If this happens the neural network is not able to undo the too large weight updates and 'revive' the neurons, as no gradients flow will be propagated back through it during backpropagation.

Several tweaks have been suggested to prevent this exact problem, by replacing the strict cutoff while maintaining the linear relationship for values larger than the threshold. One of the suggested tweaks is called exponential linear unit 'ELU' Djork-Arne Clevert [2016], which can be seen in Figure 2.13. ELU introduces an exponential part such that the activations will not be zero and therefore overcomes the 'dying Rely' problem. Li et al. [2020c]

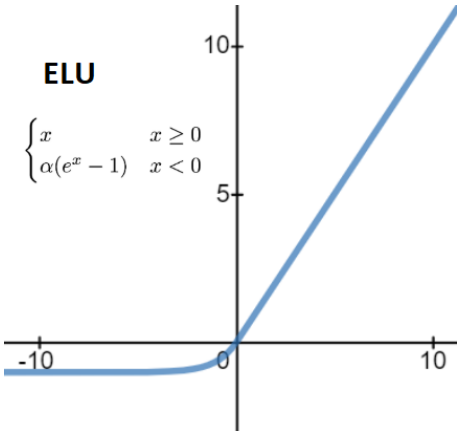

Figure 2.13: The ELU activation function. Here α is a hyperparameter controlling the curvature of the exponential curve.

2.3.3.8 Overfitting

When using a neural network there is a risk of overfitting the network to the data. Overfitting means that the model has learned to process the data it has been trained on very well, but that it cannot generalise properly and correctly handle unseen inputs. Overfitting often makes the NN useless, as the purpose of a NN is to be general and to be applicable on previously unseen data.

Overfitting can be measured by splitting the dataset up in two sets, a training set x_{train} and a validation set $x_{validation}$. Thus, during training only the training set is used to perform parameter updates. The validation set is run through the network after each epoch to see how the network performs when presented with data it has not used to train. Ideally, when no overfitting is happening, the loss of the training data is equal to the loss of the validation data: $L(x_{train}) = L(x_{validation})$ where L is the loss function. In this case, the network handles the unseen data just as well as the training data. Overfitting is visible when $L(x_{train}) < L(x_{validation})$

Overfitting is easily seen by plotting a graph of the loss function over the training epochs. An example of such a graph is shown in Figure 2.14 where the red line is the loss of the training set and the blue line is the loss of the validation set. As we can see, the validation loss is not exactly the same as the training loss, i.e. the network handles the training data slightly better. However, in practice, this is expected. The issue arises when the validation loss is much higher than the training loss, or when the validation loss starts to rise while the training loss keeps falling, as shown in Figure 2.15.

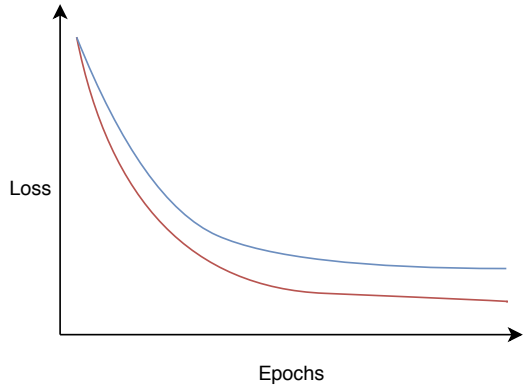

Figure 2.14: A plot of the loss function over epochs of training. The red curve shows the loss of the training data and the blue shows the loss of the validation data.

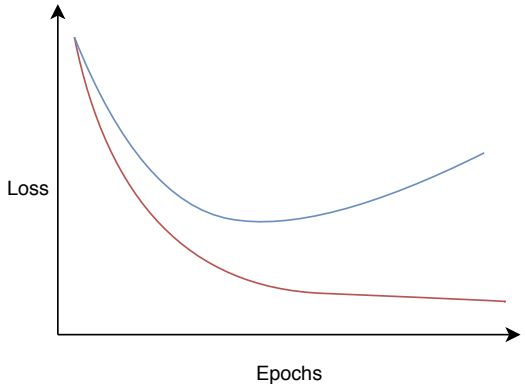

Figure 2.15: The loss of the training set (red) and validation set (blue). The validation loss deviates from the training loss indicating overfitting.

When a NN becomes overfit it essentially learns the noise in the data instead of the assumed underlying distribution of the data. A neural network is more susceptible to overfit when it has high capacity. Capacity is a measure of the space of functions the NN can represent. Increasing the layer size or the number of layers increases the capacity of the network and allows it to learn a wider range of functions as the neurons can collaborate in a more complex manner. Overfitting in the context of autoencoders can be visualised with two simplified examples, Figure 2.16 and Figure 2.17. In Figure 2.16 the loss of the training set (points in red) and the validation set (points in blue) will be about the same relative to the function sketched in black. Figure 2.17 shows strong overfitting as the function fits the training data perfectly, but is inaccurate when it comes to the validation set.

It is not a good idea to decrease the number of layers or neurons to avoid overfitting. Small networks are harder to train with methods such as SGD as they tend to have fewer local minima, but these are easy to converge to and yields a high loss. Larger NNs contain more local minima, but yield a lower loss. In practice when training a small network it is likely to converge to a bad local minimum. With a larger NN the local minima are about equally good and rely less on initialisation. [Li et al., 2020a].

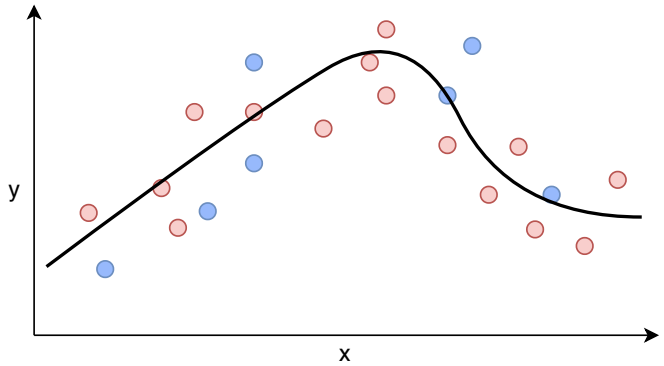

Figure 2.16: A function that has generalised from the training set (red) with no overfitting and thereby also fits the validation set (blue) well.

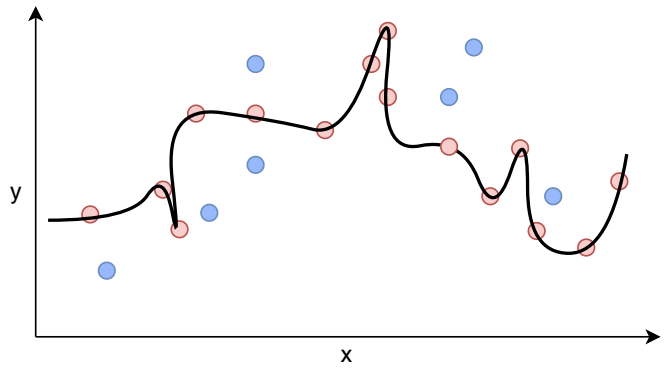

Figure 2.17: A function strongly overfit to the training set (red)

What should be done instead of decreasing the size and number of layers in the NN to avoid overfitting, is to include regularisation.

2.3.3.9 Regularisation

Regularisation constrains how much of it's capacity the NN is allowed to use and thus avoids overfitting. It essentially makes sure that the NN does not learn a "overcomplex" function that is too specific to the training data, such as the one shown in Figure 2.17. In this section, we will provide descriptions of common regularisation techniques, L1- and L2 regularisation and Dropout.

Common for both L1- and L2 regularisation is that they work by simply adding a small term to the loss function that adds extra loss when the weights misbehave. **L2 regularisation** is a constraint on every weight in the network on the form:

$$\frac{1}{2}\lambda W^2$$

Here W represents all weights of the network and λ is a regularisation parameter that controls the strength of the regularisation. L2 regularisation can thus be interpreted to penalise high weights heavily and prefer that most weights are small. The benefit of keeping the weights close to each other is to ensure that not only a few inputs dominate the output [Li et al., 2020d].

With **L1 regularisation** the term for penalising weights does not penalise high weights as hard as L2. The term for each weight in L1 is:

$$\lambda|W|$$

L1 regularisation penalises weights between 0 and 1 harder than L2 regularisation, which can make the weights in the NN become sparse, as weights are pushed towards zero. The intuition of using L1 regularisation is that the neurons end up using only the most important of its inputs and thereby become more robust to noisy inputs.

However, in practice, if you are not interested in explicit feature selection, which is provided through L1 regularisation, then L2 is generally expected to have better performance than L1. [Li et al., 2020c]

Dropout tries to simulate having many different neural networks and then take the average prediction from all the networks. Training many networks would be very computational heavy, but by simulating it through a single network with dropout, similar benefits are achieved for a fraction of the cost. Dropout works by temporarily leaving out randomly selected neurons of a layer during training making it seem like a new combination of neurons and thereby a new network each iteration of the training. Figure 2.18 illustrates how the neurons get disabled in the network during dropout. While a simple network can simulate many other networks, it has to have a certain width, i.e. the amount of neurons in the dropout layer, as it otherwise may risk becoming too thin during dropout and not able to generalise on the data. [Brownlee, 2017]

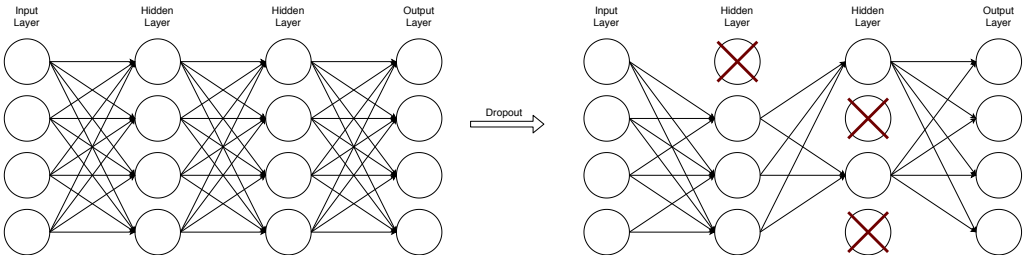

Figure 2.18: A neural network before and after nodes have been dropped out

2.3.4 Hyper parameters

As we have covered the basics of neural networks, it becomes clear that neural networks are complex functions and that they consist of many moving parts that each has its own use. Thus there is also a large number of hyper-parameters to take into account when constructing a neural network. These can be split into two categories to make them more manageable.

The first category is the "model specific hyper-parameters", which covers:

- Amount of layers
- Amount of hidden units in each layer
- Choice of activation function(s)

The model specific hyper-parameters are the fundamental structure of the neural network. The number of layers and the amount of hidden units in each layer determines the capacity of the model. The activation function on the other hand allows the introduction of non-linearity in the network, making it capable of expressing a wider range of functions.

The next category of hyper-parameters is the "optimisation specific hyper-parameters". This category covers:

- Choice of loss function
- Choice of optimiser
- Learning rate
- Choice of regulariser
- Regularisation rate
- Initialisation
- Batch size
- Number of epochs

Here the most influential hyper-parameter is the loss function as it guides the optimisation procedure and should be determined based on an understanding of the problem domain. Here we also have the influence of the regulariser which essentially can be seen as an extension to the loss function. The regularisation rate controls how much the optimisation procedure should focus on the regularisation objective.

The optimiser is the core of the learning process and controls the gradient descent algorithm, where the learning rate controls how long steps to take in the direction determined by the optimiser. Choice of optimiser and learning rate thus directly influences how fast the optimisation procedure moves, as well as the outcome of the optimisation i.e. if anything useful is learned. Choice of initialisation scheme determines the starting point of the optimisation. With an initialisation that does not suit the model well, it will not be able to learn properly and find good local minima yielding good generalisation. The batch size controls how certain the optimiser will be in the direction of steepest descent. It is hereby a trade-off parameter that will trade precision in the direction of steepest descent with how many steps that can be taken in the direction during one epoch. The number of epochs is essentially not a hyper-parameter, as it makes sense to control the number of epochs differently from model to model. Often it makes better sense to end the training based on another condition than reaching the max number of epochs e.g. when the model has converged or when it starts to overfit.

2.4 Autoencoders

As mentioned in section 2.2, an autoencoder (AE) is essentially a feedforward neural network that is trained to copy its input to its output. As we have provided the foundation of autoencoders, namely neural networks, and how they are trained, we can move on to describe how a basic autoencoder works and why an autoencoder is useful in the first place.

An AE consists of two major parts, an encoder function f and a decoder function g that are connected through a hidden layer called the *embedding layer*. The *embedding layer* is the encoded representation $h = f(x)$ of the input data, from which it should be possible for the decoder to make a reconstruction of the input $r = g(h)$. The goal of the AE is to learn to make the reconstruction, r, resemble input x as much as possible.

Autoencoders are particularly interesting in the context of clustering, and metagenomic binning when some sort of constraints are introduced to the AE that forces it to have an

interesting embedding. For example, adding the constraint that the latent layer is smaller than the input layer makes the AE learn an encoding of the input with fewer dimensions. As the AE is trained to learn to reconstruct the input from the given encoding, then the AE is forced to only contain the most salient features from the input in the latent layer.

An AE with a latent layer smaller than the input layer is called an undercomplete autoencoder. In the context of clustering and binning, the hope is that such an AE learns to find a meaningful embedding space (i.e. meaningful encodings), that helps to discriminate between the datapoints more easily than the original data and thereby lead to better clustering. In the context of clustering it is not a necessity that the encoding must be of lower dimensionality than the original data, but it makes sense how such a constraint enforces meaningful features. What matters is that the encoder has found a "good" representation of its input that allowing for better clustering.

The learning process of an autoencoder can be expressed as a minimization problem of the form shown in Equation 2.10, where L is a loss function that gets lower when the output, $g(f(x))$, is similar to the input x. L can thus be a loss function such as MSE or MAE. An autoencoder learns with the same techniques as a normal feedforward neural network.

$$L(x, g(f(x))) \tag{2.10}$$

In the remaining part of this section, we will take a look at different types of autoencoders that can be used to find an embedding that is useful for clustering contig representations. subsection 2.4.1 describes the undercomplete autoencoder in more detail. In subsection 2.4.2 we describe sparse autoencoders.

2.4.1 Undercomplete autoencoder

As mentioned in the previous section, the undercomplete autoencoder is simply just an autoencoder in which the embedding layer has fewer dimensions than the input layer. The point of an undercomplete AE is to be able to learn an efficient encoding that still maintains enough information for the decoder to reconstruct the input.

The undercomplete AE is conceptually similar to PCA. An undercomplete AE with only linear activation functions, using the mean squared error as loss function, will learn to encode inputs into the same space as PCA. For such an AE, the embeddubg layer with n dimensions will thus output an encoding similar to a PCA transformation in n dimensions. However, when non-linearity is introduced in the encoder and decoder, a more powerful nonlinear generalisation of PCA can be learned [Goodfellow et al., 2016, sec. 14.1].

2.4.2 Sparse autoencoder

Another constraint that can be applied to an autoencoder and can lead to good feature extraction capabilities is sparsity. The sparsity constraint works by constraining how many neurons in the embedding layer that are allowed to be active and give output other than zero. An autoencoder with this constraint is called a sparse autoencoder. The intuition of this constraint is, that if the embedding layer is only allowed to use a few neurons,

the layer's output has to matter a lot for the reconstruction of the input. *"If you could speak only a few words per month, you would probably try to make them worth listening to"* [Géron, 2019].

The constraint is enforced as an extra term to the loss function, just like $L1$ and $L2$ regularisation. The constraint can be added by minimising the KL divergence to a target distribution for the number of active neurons. The activations in the latent layer are measured by the mean activation of each neuron over a batch. This requires a batch of a certain size such that the mean is representative of the dataset. When the mean activation is acquired for neuron j in the latent layer, $\hat{\rho}_j$, the KL divergence is calculated between $\hat{\rho}_j$ and ρ and summed up using Equation 2.11. Here ρ is the target distribution, or the sparsity parameter, that $\hat{\rho}_j$ should take. [Ng, 2011]

$$\sum_{j=1}^{J} KL(\rho||\hat{\rho}_j) = \sum_{j=1}^{J} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \tag{2.11}$$

Commonly to control the strength of this sparsity constraint a weight is also introduced similar to the regularisation parameter λ in $L1$ and $L2$ regularisation. The weight ϕ is simply multiplied to the constraint:

$$\phi \sum_{j=1}^{J} KL(\rho||\hat{\rho}_j)$$

The sparsity constraint will not force all activations coming out of a neuron to be equally small, as it is the mean that is constrained. Some activations can be large while others can be small, and when the inputs to the autoencoder are different from one another, the activations would also be different as the important features would differ.

2.5 Clustering

In this section, we are going to cover two clustering algorithms that are suitable for metagenomic binning.

The first method is the "iterative medoid clustering" algorithm used in the binner VAMB [Nissen et al., 2018] in subsection 2.5.1, and the second is a variation of the well known DBSCAN algorithm called Hierarchical DBSCAN (HDBSCAN) in subsection 2.5.3.

2.5.1 Clustering in VAMB

The method used to form clusters in the VAMB binner [Nissen et al., 2018] uses a clustering method related to the k-medoids clustering algorithm, which is called iterative medoid clustering. Note that the algorithm described in this section is not identical to the one described in Nissen et al. [2018], the algorithm we describe here is a later adaption (VAMB 3.0) that should yield better clusters according to the authors of VAMB. The algorithm works by using a distance threshold i.e a radius, calculated from the data points/contigs

to determine if contigs should be clustered together. How this threshold is determined will be discussed later in subsection 2.5.2. Initially, the algorithm is given a *medoid radius* which is a cutoff distance. All points that are closer to the seed contig than *medoid radius* is included in an intermediate cluster. Distances between points are calculated using the cosine distance shown in Equation 2.12 where $cos(\mathbf{x}, \mathbf{y})$ is the cosine similarity between two points, $\mathbf{x}$ and $\mathbf{y}$, and n is the number of dimensions of.

$$CosineDist(\mathbf{x}, \mathbf{y}) = \frac{1}{2} - cos(\mathbf{x}, \mathbf{y}) \tag{2.12}$$

$$cos(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i}{\sqrt{\sum_{i=1}^{n} \mathbf{x}_i^2} \sqrt{\sum_{i=1}^{n} \mathbf{y}_i^2}} \tag{2.13}$$

When an intermediate cluster is determined, the mean distance from each point to the cluster seed is calculated. A new seed contig is then sampled from the intermediate cluster and a second intermediate cluster is formed. If the mean distance of the new intermediate cluster is lower than for the previous seed, the new cluster is chosen as seed contig. If the mean distance is not lower, another contig from the first cluster is selected and tested for better mean distance. This procedure is continued until all contigs in the seed cluster have been tried as seed contig, or until a threshold for futile attempts is reached without improving the mean distance to the seed. When no better seed contig can be found, the seed is used to determine the actual *threshold* value used to define a cluster. All points closer to the seed contig than the *threshold* thus forms a cluster. The cluster is removed from the dataset before a new medoid is determined.

2.5.2 Threshold

The threshold used for the iterative medoid clustering algorithm is determined from distances between the embedded contig representations. A seed contig is randomly chosen from all embeddings, and the cosine distance, Equation 2.12, between the seed and all other embedded contigs are calculated.

Each distance is then assigned to one of multiple equally large bins such that distances that are almost similar are assigned to the same bin. The bins thus form a discrete distribution of distances. The distribution is smoothed using a Gaussian kernel before a *threshold* value is determined from it. The threshold is the first local minimum of the distribution after the first local maximum, as shown in Figure 2.19.

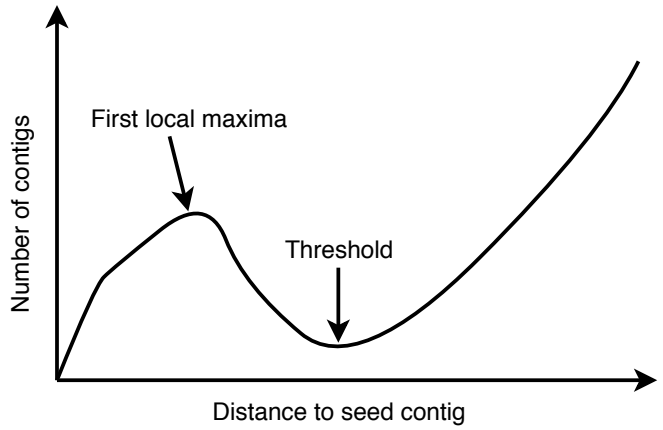

Figure 2.19: The threshold for a seed contig is defined as the first local minimum after the first local maximum.

If no threshold is found using the method described above, the threshold is set as the first point where the density was 0.8 times the maximum density and where fewer than 2,500 contigs were closer than the threshold. If none of the methods yields a threshold, the threshold of the seed is undefined and ignored. The procedure of finding threshold values is performed for 2,500 seed contigs, and the mean threshold value is determined as the threshold for the iterative medoid clustering algorithm.

The intuition of the *threshold* value and the algorithm is that it should estimate the density of data points relative to the seed contig and return a cluster that is dense, relative to the other remaining data points.

The algorithm has a range of parameters to adjust multiple details such as determining the granularity of the discrete distributions, defining a peak in the distribution, etc. Covering the effect of each parameter of the algorithm is left out of the report for reasons of simplicity and limiting tediousness.

2.5.3 Hierarchical DBSCAN

Hierarchical DBSCAN (HDBSCAN) [McInnes et al., 2017], is an algorithm that has some similarities to DBSCAN. The main difference is that the HDBSCAN algorithm creates a hierarchy of clusters that allows us to find a "non-flat" clustering. In DBSCAN the clustering is "flat". This means that HDBSCAN does not expect all clusters to have the same density. We will get more into details of this difference later in the section describing how the HDBSCAN algorithm works.

The HDBSCAN algorithm works in five steps:

1. Transform the space according to the density
2. Build the minimum spanning tree
3. Construct a cluster hierarchy/dendrogram of connected components
4. Condense the cluster hierarchy based on minimum cluster size
5. Extract the stable clusters from the condensed tree

The first step in the HDBSCAN algorithm is to estimate densities at each point to have an initial estimate of which points are noise, and which are part of a cluster. The estimate is called the "mutual reachability distance" and is shown in Equation 2.14. Here $core_k(a)$ is the distance to the kth nearest neighbour of a, and $d(a, b)$ is simply the distance between a and b.

$$d_{mreach-k}(a, b) = \max\{core_k(a), core_k(b), d(a, b)\} \tag{2.14}$$

This measure can easily be interpreted in a sketch example shown in Figure 2.20, where $k = 4$. We can see that the mutual reachability distance $d_{mreach-k}(a, b) = core_k(a)$ and is quite large as the local density around a is low. On the other hand, we have that $d_{mreach-k}(c, d) = d(c, d)$ which is much lower as c and d are close to each other and are both located in a dense area.

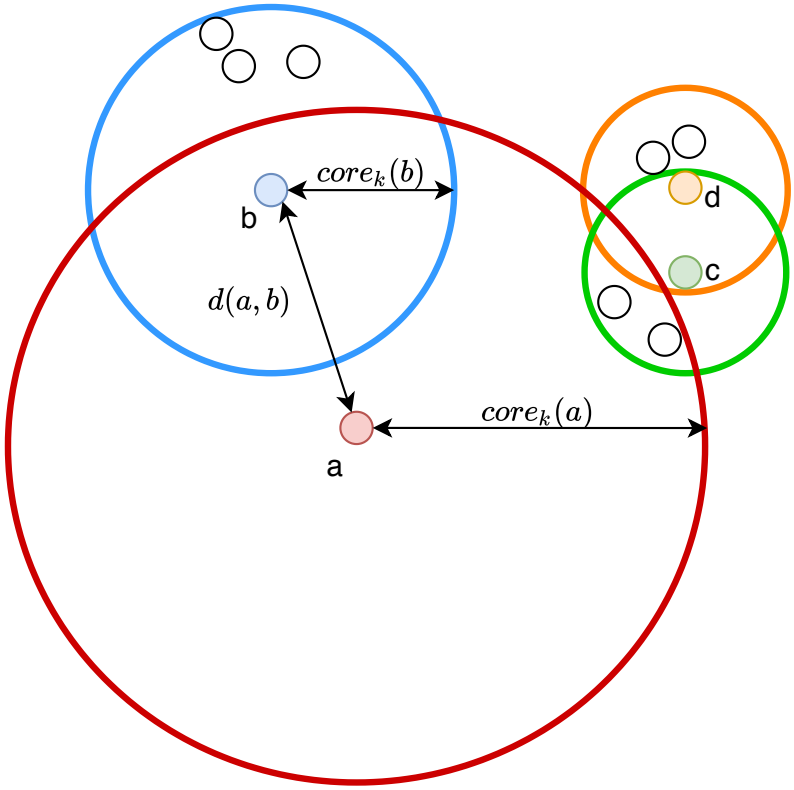

Figure 2.20: Example showing how the local reachability distance is determined and outlining how the density around points influences the distance

Once the mutual reachability distances are determined between all points, the next step is to find the minimum spanning tree to determine at which density level each point is connected to a cluster. The minimum spanning tree is constructed with a simple algorithm such as Prim's algorithm. We can see this process as initially having all points as unconnected small clusters. When an edge is added to the tree the affected vertices forms a cluster.

This leads to the next step of the algorithm, which is to build a dendrogram from the tree, which is now a trivial task. The dendrogram produced for a fictional example is shown in Figure 2.21. Here the clusters are colour coded in accordance with the number of points in the cluster. The dendrogram thus visualises the hierarchy of clusters

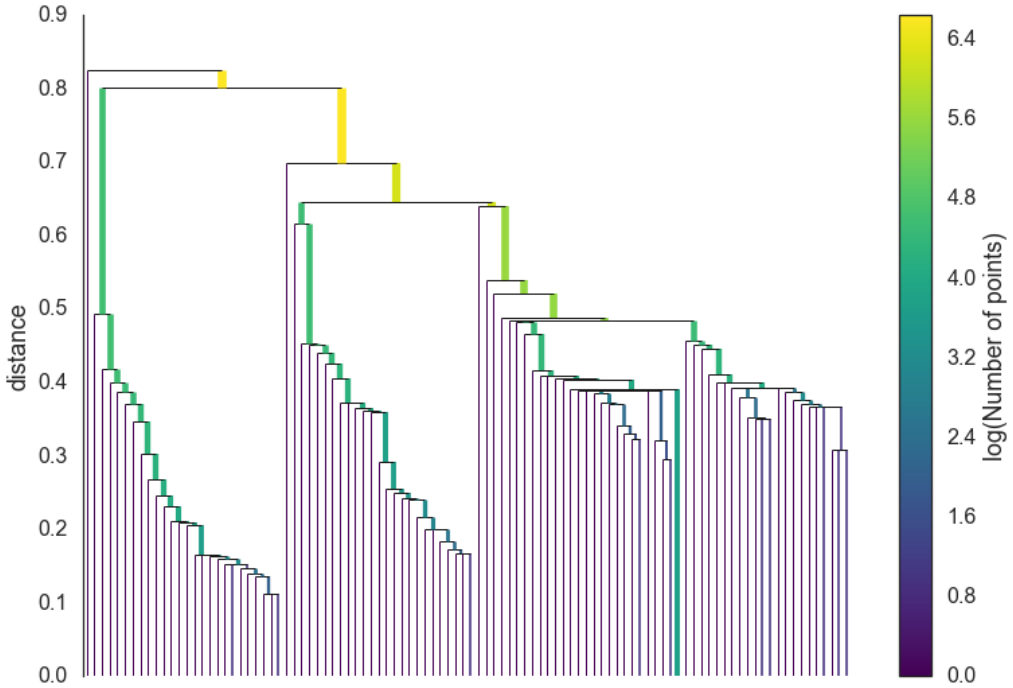

Figure 2.21: A dendrogram produced by the HDBSCAN algorithm showing the hierarchy of clusters. Image is from McInnes et al. [2018]

The ordinary DBSCAN would now essentially just draw a horizontal line through the graph corresponding to the ϵ value and be done. This is what was referred to as a "flat" clustering earlier. However, HDBSCAN wants to find clusters with varying densities (non-flat) i.e. make multiple cuts of the dendrogram at different distances (the y-axis). To determine the cuts, the parameter $minimum\_cluster\_size$ is used to reduce the dendrogram to something more comprehensible in its fourth step.

The dendrogram is traversed from top to bottom meaning that initially all points are part of the same big cluster. The distance is then gradually lowered corresponding to disconnecting components of the minimum spanning tree and thereby creating multiple smaller but denser clusters. Each time a cluster is broken into smaller clusters, the size of the clusters is evaluated. If one of the new clusters is smaller than $minimum\_cluster\_size$ it is marked as "falling out of the cluster".If more than one cluster has a size larger than $minimum\_cluster\_size$ we have more than one "actual" clusters in a split. All the actual clusters will persist in the reduced tree, and finally give a reduced dendrogram as shown in Figure 2.22. Here the y-axis is measured with $\lambda = \frac{1}{distance}$. In this figure, we can see the different persistent clusters and their number of points. We can then see at which λ value points fall out of the clusters, indicated in the figure when the cluster areas narrow down as λ increases.

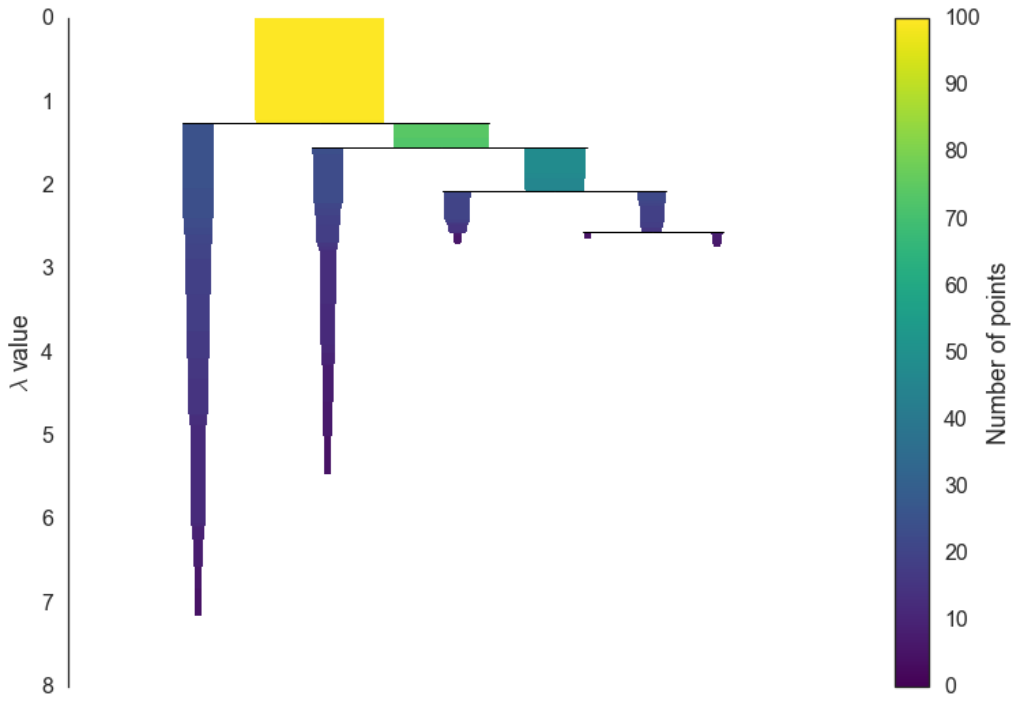

Figure 2.22: A reduced dendrogram produced by traversing the dendrogram in Figure 2.21. Image is from McInnes et al. [2018]

The final step of the algorithm is to choose a number of clusters from the reduced dendrogram. HDBSCAN measures how long each cluster has persisted, which corresponds to the size of the coloured areas in Figure 2.22. HDBSCAN keeps track of λ_{birth} and λ_{death} for each cluster. λ_{birth} is when a sub-cluster split from its parent and became its own, and λ_{death} is when a cluster is split up in two clusters. In the figure we can see that the light green cluster, has $\lambda_{birth} \approx 1.2$ and $\lambda_{death} \approx 1.6$. There is also kept track of λ_p for each point in the dataset, which is when the point falls out of a cluster.

These values are used to calculate the stability for each cluster:

$$Stability(cluster) = \sum_{p \in cluster} \lambda_p - \lambda_{birth}$$

Using this measure of stability, the reduced dendrogram is traversed from the leaf clusters and up to the root. Before considering any non-leaf clusters each leaf cluster is "selected". When considering a parent cluster a comparison of stabilities is made. If the parent's stability is lower than the sum of its children's stabilities, the parent's stability is reassigned to be the sum of the children's stabilities and the children remain "selected". If however, a parent cluster has higher stability than the sum of its children, then the parent cluster keeps its own stability value and is "selected" instead of its children. This procedure is continued up to the root node, and lastly the selected clusters are returned as the final clusters. All points not in one of these clusters are marked as noise. One can simply think of this process as finding the clusters of highest certainty i.e. the clusters persists for longest without losing too many points and without being split into smaller clusters.

2.6 Deep clustering

We have previously provided a foundation of clustering in general and of different clustering algorithms in [Linnebjerg et al., 2019]. We have also covered how neural networks work in section 2.3 as well as autoencoders in section 2.4 and how these methods can be used for feature learning. We can now move on to describe how deep learning and clustering methods can be combined to form "deep clustering" methods.

Specifically, deep clustering methods are clustering methods that use deep neural networks (e.g. autoencoders) to learn representations that are well suited for clustering. The objective of deep clustering methods consists of two parts, to minimise a network loss L_n and a clustering loss L_c. The network can essentially be any type of neural network that does meaningful feature learning. However, as we intend to only use variations of autoencoders, we will denote the network loss as L_{rec} as it is a reconstruction loss. The losses are often weighted by a hyper-parameter as shown in Equation 2.15. The network loss L_{rec} can thus be the reconstruction loss used in autoencoders, and the clustering loss L_c can be a measure of cluster densities, k-means loss, locality-preserving loss, or something similar. As an example, when using k-means clustering, the clustering loss will be the sum of distances from points in a cluster to the centroid of the cluster. [Min et al., 2018]

$$L = \lambda L_{rec} + (1 - \lambda) L_c \tag{2.15}$$

The clustering loss encourages the data points to form groups or to be become discriminative, depending on the chosen clustering loss. The network loss enforces that the learned representation should be meaningful e.g. if using an autoencoder, the model should still be able to reconstruct the input. Thereby the network loss also helps in avoiding trivial solutions for the clustering i.e. all points grouping up near the same point and forming a single dense cluster. [Min et al., 2018]

The architecture of a deep clustering model based on an autoencoder is shown in Figure 2.23.

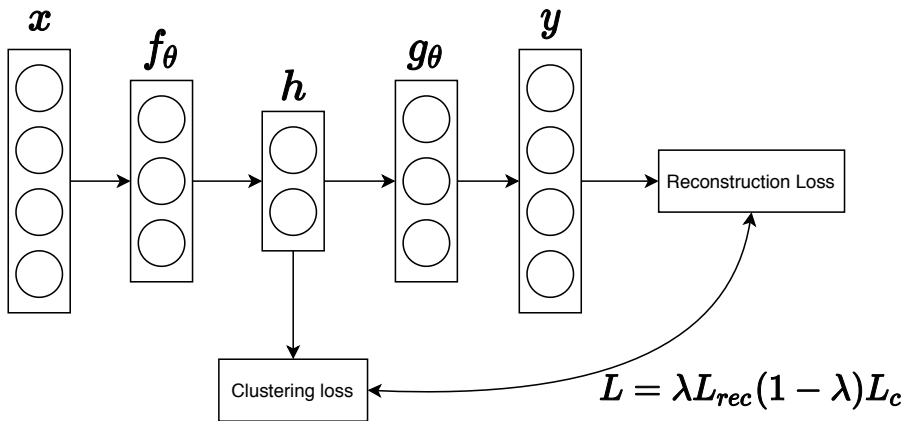

Figure 2.23: A deep clustering model based on an autoencoder. Here $\mathbf{x}$ is the input data to be clustered. The autoencoder consists of an encoder f_θ and decoder g_θ. The Loss function L is the combination of the reconstruction loss and clustering loss L_{rec} and L_c. Clustering is performed on the encoded representations $h = f_\theta(\mathbf{x})$

The idea of combining an autoencoder with clustering is thereby fairly simple. What it comes down to is essentially adding an extra term to the loss function of the autoencoder. However, there are some underlying difficulties. As mentioned there is first the issue of finding trivial solutions which are regulated with the parameter λ. However, determining the clustering algorithm can be an entire study on its own. In Min et al. [2018], which discusses multiple different deep clustering methods, most methods are using fairly simple clustering algorithms, and especially k-means clustering. Using a relatively light-weight clustering method is important in deep clustering as the parameter updates of the autoencoder relies on the clustering results. Many of the methods described in Min et al. [2018] applies deep clustering methods on datasets with a known number of clusters suitable for k-means clustering such as the MNIST dataset. However, it is quite a large limitation that the number of clusters should be known beforehand. Further, it does not match our need for a clustering algorithm, as we do not know the number of clusters. Further discussion of deep clustering and how it can be applied for metagenomic binning is provided in chapter 3.

Design and initial experiments 3

For the remaining part of the project, the goal will be to create a binner that when given appropriate input files will create the abundance and composition features and run the binning process. The binner should thereby extract the abundance and composition features from the input files and create contig representations. When the contig representations are made, the main task of binning is performed and outputs files with contigs assigned to bins.

In this chapter, we go through the design of our binner, which we name ***AutoBinner***, regarding how the preprocessing is handled in section 3.1, the implemented autoencoders in section 3.2, the choice of clustering algorithm in section 3.3, as well as a description of how we combine the reconstruction objective of the autoencoder with a clustering objective to form a joint optimisation procedure in section 3.4.

Following the design of AutoBinner we conduct a series of experiments on two different datasets CAMI Medium and CAMI High. To evaluate the bins created during the initial experiments we use metrics commonly used in metagenomic binning. These metrics are produced by the evaluation tool AMBER, by Meyer et al. [2018] and are further described in section 3.6.

To get a notion of "good bins", we contacted Prof. Mads Albertsen from AAU and were informed that what matters is the number of "high quality" bins. In short, a high quality bin is a bin that has high completeness (high recall) and low contamination (high precision). AMBER lets us perform an evaluation against ground truth bins and outputs results in a well-arranged format describing, the bins found as well as their quality.

3.1 Preprocessing

The input for metagenomic binners that uses composition and abundance features for binning is commonly a FASTA file containing the contigs to be binned alongside a number of BAM files that contain mapping information between the contigs in the FASTA file and the multiple read files the contigs originate from. We choose to follow this format to increase the ease of use of AutoBinner. Due to FASTA and BAM files being the standard input for binners, there already exist multiple modules that extract the composition and abundance features from these files. We, therefore, choose to use the implementation from VAMB Nissen et al. [2018] to extract the composition features and abundance which will be the input to our autoencoder.

3.2 Feature learning

As mentioned earlier, successful feature learning finds an embedding space that can describe the data in a more meaningful way than that of the input features. Another study [Nissen et al., 2018] has shown that feature learning on the composition and abundance features using a variational autoencoder can learn better features for clustering and subsequently better metagenomic binning results. We will, therefore, explore whether other types of autoencoders can also learn useful features that can lead to high quality bins. However, as described in section 2.4 there are different kinds of autoencoders that are capable of learning features in different ways. Specifically, we focus on two different types of autoencoders namely the basic autoencoders (AE) and the sparse autoencoders (sparse AE). To get a better understanding of how the autoencoders perform we conduct a series of experiments and tests in chapter 4.

3.3 Clustering algorithms

Putting the autoencoders aside we must choose a clustering algorithm suitable for the problem of binning. In Linnebjerg et al. [2019] we described various clustering algorithms where one of the main points was the inherent problem within the domain of metagenomic binning, that the number of clusters of course is unknown. One of the clustering algorithms where the number of clusters should not be specified is DBSCAN, however, due to its fixed ϵ parameter, it is not suitable for finding clusters of varying densities, which we expect the clusters to be.

In section 2.5 we described the clustering algorithm used in VAMB and also explained the clustering algorithm HDBSCAN. Similar for both is that they estimate local cluster densities and thereby do not suffer from the same problem as DBSCAN. Both clustering algorithms seem like good candidates to be used for binning, however, we choose to use the HDBSCAN algorithm. HDBSCAN has the property that it estimates the density in all regions of the dataset using a dendrogram before splitting the dataset into clusters. The density dendrogram is used to determine the stability of each cluster and monitor at which density the points "fall out of a cluster" and thus only extracts the most stable clusters.

3.4 Joint optimisation of clustering and reconstruction loss

In this section we will discuss how we can integrate a clustering loss for HDBSCAN which then can be optimised jointly with the reconstruction loss of the autoencoder:

$$L = \lambda L_{rec} + (1 - \lambda) L_c$$

3.4.1 Clustering loss

As the HDBSCAN does not have an inherent objective function such as lowering a loss, we choose a simple way to incorporate this. When clusters are reported by HDBSCAN we compute the centroid for each cluster. We then calculate the error by summing up the distance from the points in the cluster to the centroid. In addition, there is also the case of outliers. We could simply ignore the outliers, let their loss be zero, and let the

actual clustered points lead the optimization. Another simple solution could be to assign each outlier to their closest centroid and compute the loss as the distance to the centroid. With this method, depending on the distance measure, the points far away from a centroid would contribute more to the loss function

Inspired by Xie et al. [2016] and Chen et al. [2017], we choose to take a more elegant third solution where the points contribute to the loss function based on their distance to the found centroids. We modify the method in Chen et al. [2017] slightly such that points assigned to a cluster are weighted based on the density of nearby clusters. The idea is that a point should not be assigned to a cluster purely based on it having the shortest distance to the centroid of that cluster, but it should also consider how dense the cluster is. We see this as a way of preventing outliers to contaminate well defined clusters

The clustering loss is thus shown in Equation 3.1. Here y_i is the point currently considered, c_j is the centroid of y_i's cluster (or the closest centroid), and C is the set of all centroids. The left exp term is a gaussian kernel weighing the distance calculation, where σ_j^2 is the variance of cluster c_j. We estimate σ^2 for each cluster by setting it to the variance of the points in the cluster.

$$L_c = \exp\left(\frac{-||y_i - c_j||^2}{2\sigma_j^2}\right)||y_i - c_j||^2 \tag{3.1}$$

We assign each outlier y_i^o to a cluster using Equation 3.2. Outliers are thus only assigned to a cluster if the density within the cluster allows it.

$$\min_{c_j} \sum_{j=1}^{C} \exp\left(\frac{-||y_i^o - c_j||^2}{2\sigma_j^2}\right) \tag{3.2}$$

The intuition behind this choice of clustering loss function is that each of the centroids will pull points towards it. The underlying assumption is that the centroids found in the early iterations are mostly correct and that it is more likely that an outlier belongs to a nearby sparse cluster than to a nearby dense cluster.

3.4.2 Optimisation

Putting all things together, the last thing we need to consider is how the actual optimization algorithm will proceed. The first step will be to pretrain the autoencoder without including the clustering loss, which is commonly done in the literature [Min et al., 2018]. The pretraining allows the autoencoder to get tuned in on the data and determine an embedding space suitable for reconstruction. The assumption here is also that the embeddings learned from the pretraining will be a decent starting point for optimisation with clustering.

When the clustering loss is incorporated in the model, the optimisation will train on the entire dataset instead of mini-batches. Running the clustering on mini-batches would not be suitable for clustering as densities would vary too much between batches. Additionally, the clustering algorithm run at each epoch is by a large margin the slowest part of the optimisation procedure, and it would, therefore, be wasteful to only estimate gradients from a mini-batch instead of the entire dataset.

3.5 Datasets

In this section we describe the CAMI benchmarking datasets we use for testing, tweaking, and validating of AutoBinner. We choose to work with these datasets as there is ground truth available for each, allowing us to easily evaluate the binning with AMBER. As mentioned, we use three CAMI datasets, namely CAMI High, CAMI Medium, and CAMI Airways. The differences between the datasets are summarised in Table 3.1. The three datasets are of different complexities, i.e. the type of microbial community they contain and the number of samples they contain. Each dataset will thereby have a different number of features describing a contig, as the feature vector is a vector of length $136 + s$ where s is the number of samples.

	CAMI Medium	CAMI High	CAMI Airways
Num samples	2	5	10
bp per samples	15 Gbp	15 Gbp	5 Gbp
bp of all samples	30 Gbp	75 Gbp	50 Gbp
Num genomes	132	596	639
Number of features	136 + 2	136 + 5	136 + 10

Table 3.1: Summarisation of the three used CAMI datasets

We will use CAMI High and Medium for initial experiments and to tune parameters. We then use CAMI Airways to test our model and compare it with other binners in section 4.3.

3.6 AMBER

During training, we cannot measure the embedding quality with respect to the binning objective by monitoring the model's training and validation loss alone. We, therefore, evaluate the learned embedding with regular intervals by passing the current embedding to the clustering algorithm and evaluating the clustering with the evaluation tool AMBER.

Assessment of Metagenome BinnERs (AMBER) is an evaluation tool created by the same group that is behind the CAMI datasets; MICROBIOME Community of Special Interest. It was developed as part of a project for not only creating an evaluation tool for binners, but also for creating standard evaluation metrics within the field of metagenomic binning. The metrics were selected in a community-driven effort, and AMBER was then developed and maintained by Meyer et al. [2018]. The metrics AMBER reports are calculated by two means; bps (base pairs) and sequences (contigs). Here basepairs refer to the nucleotides that have been assigned to some bin, while it for sequences is based on the contigs that were assigned some bin.

In order to evaluate the quality of a bin, AMBER first determines which ground truth bin that matches the predicted. AMBER does this by going through all the predicted bins, and for each of them find the most similar ground truth bin which is used to evaluate.

The metrics produced by AMBER are split into two categories, one for the metrics that cover how well the binner created the individual bins, disregarding unbinned contigs. The other category covers how well the binner has created the bins with regards to the entire dataset, where here the bins are weighted based on the size of the contigs.

As we are mainly focused on finding high quality bins, we use the following metrics from AMBER to evaluate the binners. For a full list of the metrics that AMBER produces, and how they are defined, see section A.1.

- **Completeness**
  Also known as recall or sensitivity, completeness measures how many of the contigs that should be in a bin, have been placed in that bin. Completeness is calculated as follows
  $$Completeness\ r = \frac{TP}{TP + FN}$$
  where TP is the true positives, i.e. the bps that are present in a bin that also should have been there, and FN is the false negatives, i.e. the bps that should have been in a bin, but was put elsewhere.
- **Contamination**
  Contamination is the opposite of purity, i.e. precision, and is a measure that indicates how many contigs that have been placed in a certain bin, that should not have been there. Contamination is calculated as follows
  $$Contaminationc = 1 - \frac{TP}{TP + FP}$$
- **Accuracy**
  The accuracy of the bins is measured similar to purity but over the entire dataset, including the bps that were not included in any bins. The accuracy is calculated as follows
  $$Accuracy\ a = \frac{\sum_{x \in X} TP}{U + \sum_{x \in X} TP + FP}$$
  where U is the set of contigs that have not been put in a bin by the binner, and X is the set of bins made by the binner.
- **Percentage of binned bps**
  As the name suggests, this metric measures how large a percentage of the bps that have been binned. Note that this metric does not take into account whether the assignments are correct or not.

According to Bowers et al. [2017], a high quality bin is measured by having high completeness whilst having low contamination. To be more exact, if the completeness of a bin is above 90% while the contamination is below 5%, then the bin is considered a high quality bin. AMBER reports which genomes that have been recovered, i.e. the bins produced that have been mapped to a genome, in a table of varying completeness and contamination levels. For completeness, the bins are divided into the categories: Above 50%, above 70%, and above 90%, and for contamination the categories are: Below 10% and below 5%. This table is what we mainly use in the following sections to show the result of our tests.

3.7 Initial experiments

In this section we perform initial experiments with AutoBinner. We address the effect of normalising the input data in subsection 3.7.1 where we also propose a weighting scheme for the abundance and composition features to find better bins. In subsection 3.7.2 we show that the activity regulariser works for the sparse autoencoder. To further verify that the learned embeddings makes sense intuitively, we provide visualisation of the embeddings in subsection 3.7.3. Lastly, in subsection 3.7.4 we address the incorporation of a clustering objective with the objective of the autoencoder. Based on initial tests we saw that by using the Adam optimiser converging was faster than when using SGD and we therefore choose to use the Adam optimizer with a learning rate of 0.001 while testing. In addition, we choose to use the ELU activation function based on the benefits mentioned in Section 2.3.3.7. As clustering parameters we use the default clustering parameters of HDBSCAN min_cluster_size = 4, min_samples = 4.

3.7.1 Normalisation and weighting of features

It can be meaningful to z-score normalise the input features by subtracting the mean and dividing by the standard deviation in every dimension. This ensures that each feature takes on a form that follows a normal distribution with mean zero and a variance of one, which thereby makes the features lie in the same range. These properties make the loss function easier to optimise, as good or optimal weights will also lie around the same range. [Ng, 2020]

However, normalising both the composition and abundance features yields a problem. The idea of the abundance features is that contigs that originate from the same genome will have correlated abundance values across samples. Normalising the abundance features breaks this correlation, making the abundance features less meaningful. On the other hand, there is no inherent problem in normalizing the composition features.

Recall that the abundance features sum up to one, and the composition features also sum up to one. There are 136 composition features and much fewer abundance features i.e. 2 (CAMI Medium), 5 (CAMI High), and 10 (CAMI Airways). We thereby have that in general the composition features will be much smaller, and lie in a much smaller range than the abundance features.

To increase the emphasis on the composition features we choose to normalise the composition features by z-score normalisation making them lie in a larger range. This idea is demonstrated in Table 3.2. Here we can clearly see the impact of normalising the composition features. The HDBSCAN algorithm is able to create 50 bins of the highest quality, while not being able to create any decent bin with unnormalised composition features. The second thing to notice is that HDBSCAN is doing a fairly good job of clustering on its own on the normalised CAMI Medium dataset without any embedding from an autoencoder.

Tool	Contamination	>50 % completeness	>70 % completeness	>90 % completeness
Gold standard	<5% contamination	132	132	132
HDBSCAN normalised	<5 % contamination	53	52	50
	<10 % contamination	54	53	51
HDBSCAN unnormalised	<5 % contamination	0	0	0
	<10 % contamination	0	0	0

Table 3.2: Performance of HDBSCAN on all 138 (only 2 samples) features of the CAMI_Medium dataset for both normalised and unnormalised composition features

We can also see the effect of normalising the composition features when doing feature learning with an autoencoder in Table 3.3. Once again no high quality bins are found when composition features are not normalised. The normalization of the composition features increased their range, which gave the autoencoder an added incentive to focus on reconstructing the composition features making them more prominent in the embedding.

Tool	Contamination	>50% Completeness	>70% Completeness	>90% Completeness
Gold standard	<5%	132	132	132
AE + HDBSCAN Normalised composition	<5%	47	45	40
	<10%	50	48	41
AE + HDBSCAN Unnormalised composition	<5%	0	0	0
	<10%	0	0	0

Table 3.3: The effect of normalising the composition features before encoding the features on the CAMI Medium dataset. After encoding, HDBSCAN is run to produce the clustering.

The results in Table 3.2 and Table 3.3 despite not being comprehensive, do not show an indication that clustering on the embedding is beneficial. However, we see a potential cause of why the embedding was not useful. According to Albertsen et al. [2013], Kang et al. [2019], and Nissen et al. [2018] (among others) the abundance features are more defining for genomes than the composition features, especially when multiple samples are available. As we normalized the composition features, we also made them more influential (implicitly making the abundance features less influential) leading to worse binning results. Therefore we introduce a simple weighting scheme for the loss of the autoencoder that favors the reconstruction of abundance features.

The weighting scheme is given in Equation 3.3, where $COMP_{error}$ is the reconstruction error of the composition features, ABD_{error} is the abundance reconstruction error and s is the number of samples in the dataset. That the weighting is scaled according to the number of samples resonates nicely with the fact that the more abundance features there is, the more reliable they are [Kang et al., 2019], [Nissen et al., 2018].

$$L_{rec} = \frac{COMP_{error}}{s} + ABD_{error} \tag{3.3}$$

When running the test again, but now with the weighting scheme, the results shown in Table 3.4 are achieved. From the results, it can be seen that the weighting results in a major improvement in the number of high quality bins found. We see this as supporting the claim about the importance of the abundance features in the context of metagenomic binning as well as showing the potential of combining HDBSCAN and autoencoders.

Tool	Contamination	>50 % completeness	>70 % completeness	>90 % completeness
Gold standard	<5% contamination	132	132	132
HDBSCAN + $AE_{weighted}$	<5 % contamination	74	73	70
	<10 % contamination	76	75	72

Table 3.4: Performance of HDBSCAN when combined with an undercomplete stacked autoencoder.

In order to verify that the autoencoder is favouring reconstructing the abundance features, we perform a sensitivity analysis, i.e. computing the gradient of the loss with respect to each dimension of the input. This is done after having trained the autoencoder for 100 epochs and then feeding the whole dataset to the autoencoder and summing the aforementioned gradients. The results can be seen in Figure 3.1 and Figure 3.2, which clearly shows that the autoencoder is most sensitive regarding the abundance features.

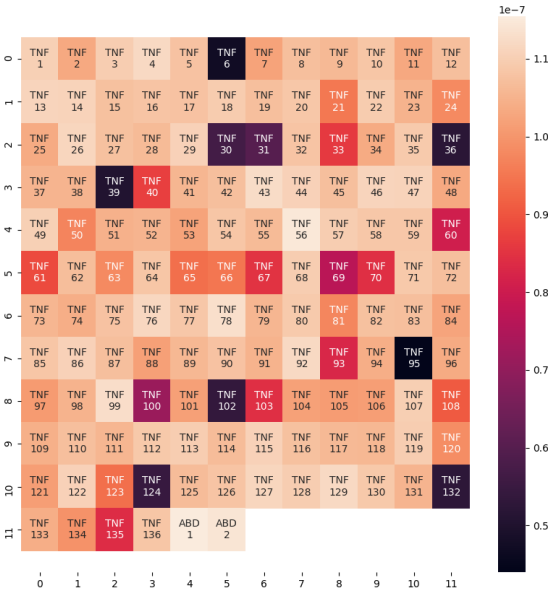

Figure 3.1: Sensitivity analysis - Unweighted loss

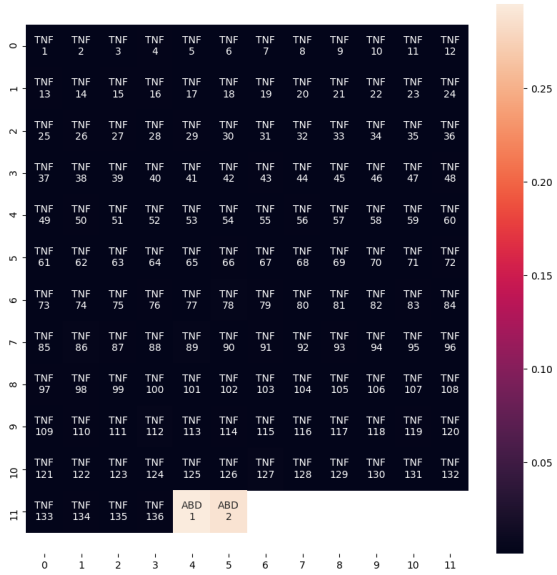

Figure 3.2: Sensitivity analysis - Weighted loss

So far we have shown that HDBSCAN can work with an autoencoder to find better bins, and that z-normalising the composition features further improves the binning. We have also argued, and shown, how the abundance features should be weighted higher than the composition features to yield a representation that works well for clustering. We have thereby covered that the pretraining of the model is meaningful.

3.7.2 Activation regularisation

For the sparse autoencoder we check whether the activation regulariser is enforcing the distribution of the activations in the embedding layer correctly by measuring the average activity of each neuron in the embedding layer over the complete dataset. In Figure 3.3 and Figure 3.4 results can be seen of enforcing the target distribution of the activations to be 0.1. A sigmoid activation function has been used on the embedding layer, such that values lie between 0 and 1.

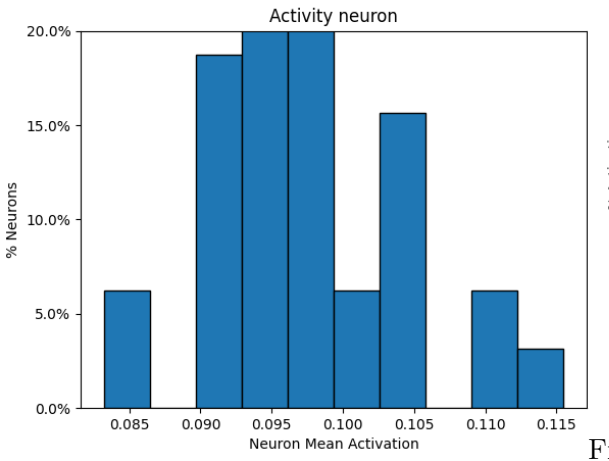

Figure 3.3: Distribution of the activations in the embedding layer, when using a sparse autoencoder having an activity regulariser enforcing the activations to be 0.1

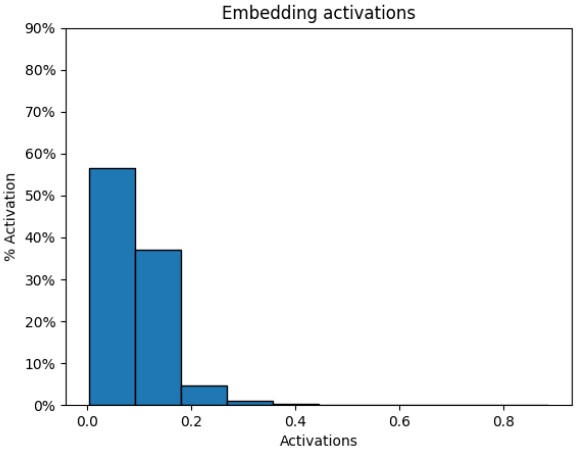

Figure 3.4: Distribution of the activations in the embedding layer, when using a sparse autoencoder having an activity regulariser enforcing the activations to be 0.1. Should be read as nearly 60% of the activations of the embedding layer have a value between 0.0 and 0.1 and so forth.

3.7.3 Visualising learned embeddings

Finally to get a better visualisation of the embeddings and their separation we take a subset of the embeddings corresponding to ten of the largest clusters from the CAMI High dataset based on the ground truth references. The embeddings are of 32 dimensions and are then projected to three dimensions using t-SNE. Similarly, we choose to perform PCA on the cami high dataset projecting the dataset down to 32 dimensions. The embeddings of 10 of the largest clusters made by PCA are similarly projected down to three dimensions by t-SNE. The visualisations can be seen in figures 3.5, 3.6, 3.7, and 3.8.

From the figures we can see that from a visual perspective at least, the ten clusters are better separated when using the embedding from the encoder than that of PCA. The embedding from PCA indicates that six or seven clusters are present instead of ten, where six of them seem well separated. The autoencoder on the other hand finds ten well defined clusters, however, a noisy area is present. The results are available at http://mi108.northeurope.cloudapp.azure.com/ allowing for better exploration of the three-dimensional t-SNE embeddings.

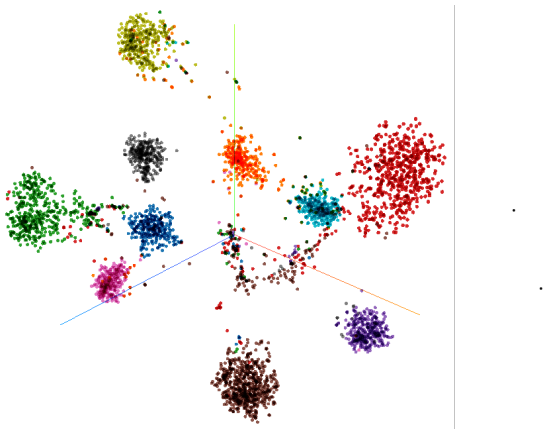

Figure 3.5: T-SNE in three dimensions of the embedding made by the encoder. The data points are coloured according to their ground truth.

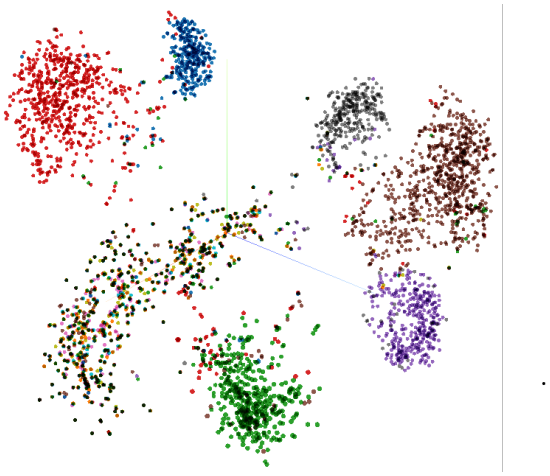

Figure 3.6: T-SNE in three dimensions of the embedding made by PCA. The data points are coloured according to their ground truth.

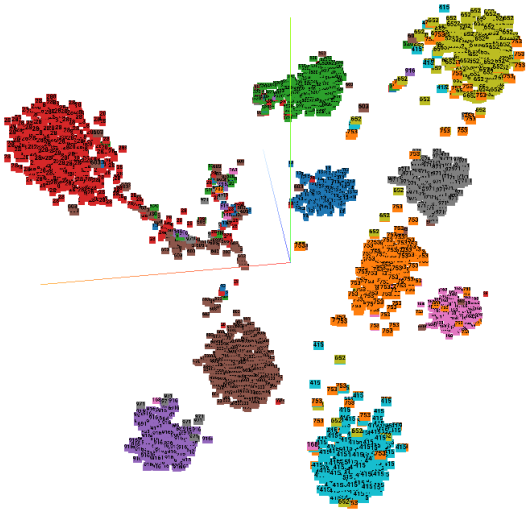

Figure 3.7: The same embedding made by t-SNE of the encoder embedding, seen from another angle. The data points are coloured and labeled according to their ground truth.

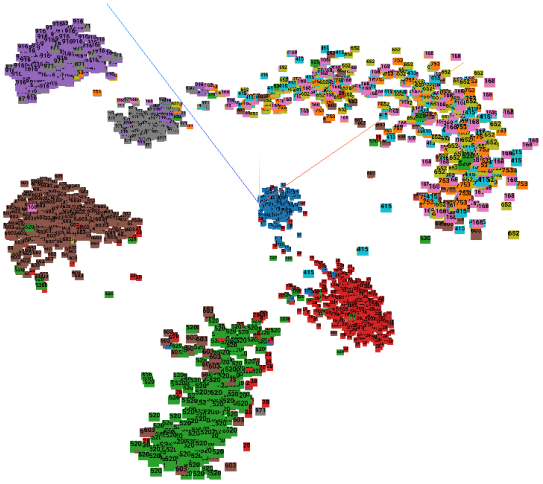

Figure 3.8: The same embedding made by t-SNE of the PCA embedding, seen from another angle. The data points are coloured and labeled according to their ground truth.

3.7.4 Deep clustering

We followed the procedure for incorporating the clustering loss in the training as described in section 3.4. To our disappointment, the inclusion of this secondary training objective did not improve our binning results. A further discussion of possible reasons why this was not successful is provided in section 5.1. We choose to proceed without having the clustering influence the training directly.

Parameter tuning and comparison 4

In this chapter, we describe the process of finding the final parameters for AutoBinner. We determine the model specific and optimisation specific parameters of the autoencoder in section 4.1. In section 4.1.1 the impact of the two parameters of HDBSCAN is tested. The weighting scheme balancing the importance of the composition and abundance features is tweaked in subsection 4.1.2. We compare AutoBinner against, VAMB, MetaBAT2 and CONCOCT which are briefly described in section 4.2.1, 4.2.2, and 4.2.3. Finally, the results of the comparison are shown section 4.3.

4.1 Tweaking hyper-parameters of AutoBinner

In this section we will find suitable hyper-parameters for the autoencoder yielding good binning results based on the CAMI High dataset. We will however, not do a full systematic test of all hyper-parameters, as such a test is very time consuming. We instead evaluate different settings of number of hidden layers, number of neurons in each layer, as well as using dropout with different drop out rates. The results can be seen in Table 4.1 and Table 4.2.

Layer size: 200		Layers		
		3	4	5
Dropout	0	453	**455**	447
	0.1	435	423	431
	0.2	428	422	424

Table 4.1: Test results of different architectures of a stacked autoencoder having 200 neurons in each hidden layer and 32 in the embedding layer. Results are based on the number of high quality bins reported by AMBER, which have a completeness of at least 90% while being at most 5% contaminated.

Layer size: 300		Layers		
		3	4	5
Dropout	0	**454**	453	446
	0.1	437	434	443
	0.2	431	434	435

Table 4.2: Test results of different architectures of a stacked autoencoder having 300 neurons in each hidden layer and 32 in the embedding layer. Results are based on the number of high quality bins reported by AMBER, which have a completeness of at least 90% while being at most 5% contaminated.

The results show that for the different combinations tested, the number of layers and the layer size do not seem to have much impact on the results. This indicates that the tested autoencoders all have the required capacity. For the results regarding dropout, we did not see any improvement, as the results indicate that the higher the dropout the fewer high quality bins are found.

For the sparse autoencoder, we saw very similar results to the results from Table 4.1 and Table 4.2, and thereby no immediate benefits of restricting the activity of the neurons in the embedding layer. We therefore choose to proceed without the sparsity constraint in the autoencoder.

4.1.1 Choice of parameters for HDBSCAN

The HDBSCAN algorithm has two main parameters $min\_samples$ and $min\_cluster\_size$ to control the clustering. To determine which combination of these are suitable for the binning domain and the encoding, we make a grid search. Initial ad-hoc experiments showed that the default settings of $min\_samples = 4$ and $min\_cluster\_size = 4$ worked fairly well, and therefore we search for parameters in said area. The tests were run on the embedding from a simple stacked autoencoder with 4 layers of size 100 with 32 neurons in the embedding layer. The reported score is the number of found highest quality bins, with contamination $< 5\%$ and completeness $> 90\%$. The results of the search are shown in Table 4.3, where the best results are shown in bold. From the table, we can see that the default parameters (4 and 4) are still doing well, but that we can do a little better with $min\_samples = 2$ and $min\_scluster\_size = 6$, underlined in the table.

		min_cluster_size								
		2	3	4	5	6	7	8	9	10
min_samples	1	401	424	431	436	435	432	425	421	414
	2	426	435	**437**	**439**	**<u>441</u>**	434	424	420	413
	3	429	432	**437**	**441**	**439**	432	422	419	410
	4	433	433	**437**	**437**	434	429	419	414	406
	5	426	432	433	433	432	426	418	415	400
	6	423	424	424	425	424	425	414	408	396

Table 4.3: Grid search for good HDBSCAN parameters. The best results are highlighted in bold. The reported score is the number of found highest quality bins, with contamination $< 5\%$ and completeness $> 90\%$

4.1.2 Weight tuning

Having established the importance of abundance features in section 3.7, we will now attempt to determine how the reconstruction loss of the two kinds of features should be weighted. We do this by modifying the weighting scheme in Equation 3.3 to:

$$L_{rec} = \frac{COMP_{error}}{s * w} + ABD_{error} \tag{4.1}$$

Here w is a parameter we control explicitly to determine how greatly we should weight the abundance features. The results of these experiments are shown in Table 4.4

Weight w	Number of high quality bins
2	456
6	457
20	465
39	467
40	**467**
41	463

Table 4.4: How different weighting of the features influences the outcome of the clustering on the CAMI high dataset. The autoencoders encoder and decoder consists of 4 layers with 200 neurons in each, and 32 neurons in the embedding layer, as this is the model performing best on CAMI High

As we can see from Table 4.4 the weighting that yields our best results (40) is fairly extreme. However, a heavy weighting is in fact in accordance with other binners' weighting schemes, such as Nissen et al. [2018] and Kang et al. [2019]. We make an additional test and run HDBSCAN on the abundance features only, which finds 391 high quality bins. From this we again see the importance of the abundance features, but also that the composition features is a necessity. To follow this tangent a bit further, we run an additional test with clustering on an embedded representation of the features. We did however not see results better than clustering on the "raw" non embedded abundance features.

To summarise, the best parameters we found for the autoencoder are as follows

- 4 layers
- layer size of 200
- 32 neurons in the embedding layer

and for the HDBSCAN the best parameters we found are

- Min cluster size of 6
- Min sample size of 2

With these parameters, we move on to comparing AutoBinner with other state of the art binners in the next chapter.

4.2 Comparison

In this section, we will present a comparison of AutoBinner, VAMB [Nissen et al., 2018], MetaBAT2 [Kang et al., 2019], and CONCOCT [Alneberg et al., 2013]. In sections 4.2.1, 4.2.2, and 4.2.3 we provide a short introduction to the binners. In section 4.3 we present the binning results on the three datasets CAMI High, CAMI Airways, and CAMI Medium. Note that CAMI Airways is the most fair dataset for the comparison as we have used CAMI High to determine hyper-parameters, which is also the case for CAMI Medium, but to a lesser extent.

4.2.1 CONCOCT

CONCOCT was the first binner to use both composition and abundance as its input features, which has since then become a commonly used representation. CONCOCT uses PCA to transform the data followed by fitting a Gaussian mixture model to the data using a method similar to the Expectation-Maximization algorithm.

CONCOCT has been cited by various binners, which is an indication that the ideas and concept behind CONCOCT has become the basis foundation for many binners. We choose to compare AutoBinner to CONCOCT as we see it as the baseline binner, and that the goal for AutoBinner is to at least be comparable to CONCOCT.

4.2.2 MetaBAT2

MetaBAT2 works by running a modified version of the label propagation algorithm allowing the number of clusters to be unknown. MetaBAT2 has a contig cutoff before running the label propagation algorithm. MetaBAT2 has a cutoff threshold of 2.5 kbp for minimum size contigs. The contigs below the threshold are assigned to the bins found by label propagation.

In our previous report we saw that not only was MetaBAT2 very fast, it also produced very good results when more than three samples were provided to it. Furthermore, in Koslicki et al. [2017] MetaBAT, the predecessor of MetaBAT2, showed good results, which was also the reason we included it in our previous report. We choose to compare AutoBinner against MetaBAT2 to have a ceiling to aim for, i.e. by having a well performing state of the art binner to compare us against, we have a way of knowing when AutoBinner is performing well.

4.2.3 VAMB

VAMB is a newer binner, and as with MetaBAT2, it is at the time of writing also still receiving refinement and tweaking (*https://github.com/RasmussenLab/vamb*). VAMB stands out from other binners by making an encoding of the abundance and composition features using a variational autoencoder before clustering the contigs into bins. The clustering of the encodings is done using the iterative medoid clustering algorithm described in subsection 2.5.1.We choose to compare AutoBinner against VAMB, as the overall structure of the binners is quite similar. Furthermore, according to the creators of VAMB, it performs better than other state of the art binners, including MetaBAT2, which also gives AutoBinner a goal to reach for.

4.3 Comparison results

Here we present the results from having run MetaBAT2, VAMB, and CONCOCT, as well as our binner AutoBinner, on the datasets CAMI High, CAMI Airways, and CAMI Medium. Once again our primary measure of binning quality will be the number of high quality bins with completeness $> 90\%$ and contamination $< 5\ \%$. However, we will also provide a more detailed view of the results with regard to other metrics. The results are also available as an interactive webpage at: http://mi108.northeurope.cloudapp.azure.com.

4.3.1 Results: CAMI High

The results of running the four binners on the CAMI High dataset are shown in Table 4.5. The metric shows that AutoBinner is in fact the best performing on this dataset with regards to all categories of bin quality. AutoBinner is capable of finding 109 more of the highest quality bins than MetaBAT2, 134 more than VAMB, and 390 more than CONCOCT. AutoBinner also performs better with regards to finding the lower quality bins with lower completeness and more contamination. However, note that the highest completeness bins (>90%) are a subset of the lower completeness bins (>50% and >70%). The same thing goes for contamination, where the lowest contamination bins (<5%) are a subset of the higher contamination bins (<10%).

Binner	Contamination	<50% completeness	<70% completeness	<90% completeness
Gold standard	0 %	596	596	596
AutoBinner	<5 %	**489**	**486**	**467**
	<10 %	**489**	**486**	**467**
MetaBAT2	<5 %	412	406	358
	<10 %	424	418	366
VAMB	<5 %	386	384	333
	<10 %	387	385	334
CONCOCT	<5 %	81	81	77
	<10 %	87	87	81

Table 4.5: A comparison of AutoBinner against MetaBAT2, VAMB, and CONCOCT on the CAMI High reporting the high quality bins

We include more comprehensive metrics of the binning results in Table 4.6. In this table, the two topmost metrics "Average purity (bp)" and "Average completeness (bp)" are metrics that are only calculated on the reported bins i.e. unbinned contigs are not contributing to these metrics. Below the grey line, all contigs are used to calculate the metrics. As we can see from the table, AutoBinner has better results in most metrics, except for average purity with 5% and 3.5% less than VAMB and MetaBAT2. AutoBinner lies almost on par with MetaBAT2 and VAMB for the percentage of binned bp and contigs, where CONCOCT has managed to bin the most. We do however see that CONCOCT also has fairly weak results for the most part. That AutoBinner falls a fair bit behind on Average Purity indicates that it could be a possible improvement to make the binner more conservative.

Metrics for assigned bins	AutoBinner	MetaBAT2	VAMB	CONCOCT
Average purity (bp)	86.6%	90.1%	**91.6%**	59.4%
Average completeness (bp)	**95.1%**	94.3%	92.1%	94.3%
Metrics for the full dataset				
Accuracy (bp)	**87.2%**	79.3%	76.2%	36.9%
Misclassification rate (bp)	**9.3%**	19.2%	20.4%	63.1%
Purity (bp)	**90.6%**	80.8%	79.6%	36.9%
Completeness (bp)	**94.9%**	94.3%	92.3%	94%
Percentage of binned bp	95.9%	97.9%	95.5%	**99.9%**
Percentage of binned contigs	65.2%	63.3%	65.6%	**87.7%**

Table 4.6: Detailed results of running AutoBinner, MetaBAT2, and VAMB on the CAMI High dataset. The best results for each metric is highlighted in bold font

4.3.2 Results: CAMI Airways

Seeing how AutoBinner got very good results on the CAMI High dataset, we try and run the binners on CAMI Airways to see how it handle a different and more complex dataset with more samples. We got the dataset from Simon Rasmussen(VAMB), where the composition and abundance already had been calculated and saved in a file. This means that we can not run CONCOCT on this dataset, as it requires the BAM files to calculate its abundance features, whereas VAMB, MetaBAT2, and AutoBinner all can take the abundance and composition files as input instead of the FASTA and BAM files. The results from running the binners on the Airways dataset can be seen in Table 4.7, where the best binning results are highlighted in bold font. Here we can see that none of the binners are able to perform as well as they did on the High dataset. The best binner, VAMB, can only find 24 (3.75%) of the 639 high quality ground truth bins. We also see that AutoBinner, unfortunately, has a drastic decrease in performance, compared to our results on CAMI High. AutoBinner finds 12 and 22 fewer highest quality bins than MetaBAT2 and VAMB respectively. AutoBinner is also far inferior with the lower quality bins.

Binner	Contamination	<50% completeness	<70% completeness	<90% completeness
Gold standard	0 %	639	639	639
AutoBinner	<5 %	6	5	2
	<10 %	7	6	3
MetaBAT2	<5 %	35	27	14
	<10 %	40	31	14
VAMB	<5 %	**83**	**70**	**24**
	<10 %	**90**	**77**	**27**

Table 4.7: Results of running the binners on the CAMI Airways dataset.

When we look into more detail of the CAMI Airways results in Table 4.8, we can see that AutoBinner generally performs worse than VAMB and MetaBAT2 for all metrics. Again we can see that AutoBinner creates bins with low purity, which was also AutoBinners biggest weakness on CAMI high. We also see that AutoBinner actually bins a fair amount

of contigs and bp. This again indicates that the clustering could be more conservative to potentially find a larger amount of high quality bins.

Metrics for assigned bins	AutoBinner	MetaBAT2	VAMB
Average purity (bp)	15.5%	74.6%	**78.1%**
Average completeness (bp)	28.1%	25.9%	**40.9%**
Metrics for full dataset			
Accuracy (bp)	15.3%	31%	**45.3%**
Misclassification rate (bp)	75.8%	51.1%	**39.9%**
Purity (bp)	24.2%	48.9%	**60.1%**
Completeness (bp)	21.5%	**56.8%**	39.1%
Percentage of binned bp	63.4%	63.3%	**75.4%**
Percentage of binned contigs	**62.1%**	39.1%	50.5%

Table 4.8: Detailed results of running AutoBinner, MetaBAT2, and VAMB on the CAMI Airways dataset. The best results for each metric is highlighted in bold font.

4.3.3 Results: CAMI Medium

As AutoBinner showed a rather poor performance on the CAMI Airways dataset, we try and run AutoBinner on CAMI Medium which is more similar to CAMI High which we used to tweak AutoBinner. The results from running the binners on the medium dataset can be seen in Table 4.9. In the table, we see that AutoBinner (with $w = 40$) was able to perform better than MetaBAT2 and VAMB, but did not perform as well as CONCOCT did. We also included our initial test results from section 3.7, to showcase that AutoBinner is able to perform better. The big difference here is that the current version of AutoBinner weighs the abundance much higher than in the initial test (by a factor of 20) As the medium dataset only has two samples, the abundance ended up having too much weight and not enough abundance features to support it. We discuss this further in section 5.2.

Binner	Contamination	<50% completeness	<70% completeness	<90% completeness
Gold standard	0 %	132	132	132
AutoBinner*	<5 %	74	74	70
	<10 %	76	75	72
AutoBinner	<5 %	46	41	32
	<10 %	46	41	32
MetaBAT2	<5 %	18	18	14
	<10 %	26	25	21
VAMB	<5 %	35	35	30
	<10 %	36	36	31
CONCOCT	<5 %	65	65	63
	<10 %	68	68	65

Table 4.9: Results of running the binners on the CAMI Medium dataset. * = Abundance weighting (w=1) as in our initial test on Medium in section 3.7

Discussion 5

5.1 HDBSCAN and deep clustering

One of the larger challenges in metagenomic binning is that the correct number of bins is unknown, making many of the common clustering algorithms unusable such as k-means. The HDBSCAN algorithm solves this problem by determining clusters based on local densities and cluster stability and thus estimating the number of clusters.

HDBSCAN does not have an inherent clustering objective like k-means that tries to optimise the placement of centroids. As we attempted to include HDBSCAN with the autoencoder for joint optimization, we took a simple solution and calculated the centroid for each cluster and use the sum of distances from the points in the cluster to the centroid as the error function. This may be problematic, as HDBSCAN can find clusters of arbitrary shapes, and centroids for such clusters could lie close to each other. During optimisation, such two different clusters would move towards each other and merge into one.

During the optimisation, the embedded points essentially move around to minimise the clustering loss, and all points move towards some centroid. However, as this happens, clusters, and individual outlier points, will inevitably merge together and form new clusters. We saw in our experiments that the number of clusters determined for each epoch was very unstable. We believe that this instability is caused by the issues described above, and is therefore the reason our model did not succeed in joint optimisation of the reconstruction and clustering loss.

5.2 Weighting of composition and abundance

During our experiments, and in our comparison of AutoBinner against other binners, we saw that the weighting of abundance had a drastic influence on the number of high quality bins found. We experimented with a simple weighting scheme that allowed us to easily adjust the weighting yet no conclusive setting was found. Using a weight $w = 40$ worked well for CAMI High, but was not suitable when running CAMI Airways. We also saw that $w = 40$ didn't work well for CAMI Medium, where instead $w = 1$ worked well. It thus seems like 40 was simply too high as only two samples, i.e. two abundance values were available in CAMI medium. We believe that the current results on CAMI Airways can be improved by a more sophisticated weighting scheme that increases the weight drastically when many samples are available.

Furthermore, despite the clear importance of abundance, the composition did still contribute to the binning results. A more sophisticated weighting scheme could allow the composition feature to have a bigger influence for long contigs, as the longer the contig the more robust/precise the composition feature.

5.3 CAMI Airways vs CAMI High

The results of running Autobinner on CAMI airways were not impressive, however, we do not see this as a reason to discard our approach of using autoencoders and HDBSCAN, as there can be multiple explanations to why we did not achieve better results. To get a better understanding of potential causes of the poor results it can be helpful to compare the CAMI airways dataset to the CAMI high dataset on which AutoBinner performed well. One of the main differences between the two datasets is the distribution of long and short reads.

In Table 4.6 showing our results on CAMI High we saw that AutoBinner were able to bin 95.9% of the bps, but only 65.2% of the contigs, which indicates that the majority of the bps in the dataset are part of the minority of the contigs, i.e. the long contigs.

Knowing this we try to change the clustering parameters of HDBSCAN to $min\_cluster\_size = 2$ and $min\_samples = 1$ and run AutoBinner on CAMI Airways again. The result can be seen in Table 5.1 and Table 5.2. Here we see a drastic increase of high quality bins found by going from 2 to finding 13 high quality bins. Furthermore, if we look in Table 5.2 we see that the Average Purity has gone from 15.3% all the way op to 61.9% and the accuracy has gone from 15.3 to 41.4%. This shows that these new clustering parameters yields a much more conservative clustering with clearly better results.

Binner	Contamination	<50% completeness	<70% completeness	<90% completeness
Gold standard	0 %	639	639	639
AutoBinner*	<5 %	21	17	13
	<10 %	22	17	13
AutoBinner	<5 %	6	5	2
	<10 %	7	6	3

Table 5.1: Results from running AutoBinner on the Airways dataset with different clustering parameters. The * refers to different clustering parameters $min\_samples = 1$, $min\_cluster\_size = 2$ instead of $min\_samples = 2$, $min\_cluster\_size = 6$

Metrics for assigned bins	AutoBinner (min_samples=2 min_cluster_size=6)	AutoBinner* (min_samples=1 min_cluster_size=2)
Average purity (bp)	15.5%	**61.9%**
Average completeness (bp)	**28.1%**	12.3%
Metrics for the full dataset		
Accuracy (bp)	15.3%	**41.4%**
Misclassification rate (bp)	75.8%	**24.8%**
Purity (bp)	24.2%	**75.2%**
Completeness (bp)	**21.5%**	12.5%
Percentage of binned bp	**63.4%**	55%
Percentage of binned contigs	**62.1%**	54%

Table 5.2: Extended results from running AutoBinner on the Airways dataset with different clustering parameters

5.4 Contractive autoencoder

In addition to the basic autoencoder and the sparse autoencoder, we also implemented a contractive autoencoder. We chose to leave it out of the report as we were not able to run sufficient testing as our implementation was too slow. This was caused by the fact that the contractive autoencoder adds a term to the loss function which involves calculating the Frobenius norm of the Jacobian matrix of the embedding with respect to the inputs to the encoder. We suspect that it is our implementation that is suboptimal as the computation was not utilising the GPU and thereby became a major bottleneck during the training. For a further explanation of contractive autoencoders and their potential benefits see section A.3.

5.5 Additional features

We have only looked at using abundance and composition as input features during this project, however, it could also be interesting to look at other features that might improve the binner. The current composition and abundance features do not provide sequential information of the contigs, i.e. the order of the nucleotides. It could be of interest to make use of the reads in form of the raw sequence of nucleotides A,C,T and G. Here a Recurrent Neural Network(RNN) or a Convolutional Neural Network(CNN) could be of use.

5.6 Datasets

We chose to use the CAMI datasets as they allowed us to evaluate AutoBinner in a conventional manner, as we had ground truth binning references for all contigs. The ground truth references are available as the datasets are synthetically generated by using contigs from a catalogue of genomes. This has the drawback that the datasets might not reflect real metagenomic samples perfectly.

One approach that allows validating on non synthetically generated data, i.e. data without ground truth references, is to use the tool CheckM by Parks et al. [2014]. CheckM uses its own database of known "marker genes" to estimate the completeness and contamination of the bins. We chose not to pursue this approach during the project, as our lack of knowledge within the domain that CheckM uses, would make the validation less transparent.

Conclusion 6

In this project, we set out to create a tool for metagenomic binning that uses deep learning to learn well-suited contig representations for clustering with our chosen clustering method.

We created the binner AutoBinner that combines the embedding made by an autoencoder with the clustering algorithm HDBSCAN. We have through experiments showed how different factors impact the binning results, such as the architecture of the autoencoder and the weighting of the composition and abundance features. We evaluated our progress through multiple experiments to test whether the autoencoder learned as expected, including sensitivity analysis on the input, and measuring the activity of the neurons in the embedding layer for the sparse autoencoder. In addition, to get a better visual understanding we used t-SNE on the embedding created by the autoencoder and compared it to an embedding created by using PCA.

Our experiments showed that the autoencoder did learn an embedding that was well suited for both reconstruction, as well as for clustering contigs, and thereby improved the binning results. Our experiments further showed the importance of the balance between the abundance and composition features, and how this balance strongly influences the binning results. Therefore, we proposed a simple weighting scheme to finetune this balance, however, additional testing is required to determine a weighting suitable for multiple datasets.

We attempted to integrate the clustering process in the autoencoder optimisation process to form a deep clustering algorithm and make the autoencoder learn a feature representation matching the clustering objective and yield a better clustering. However, we did not succeed in making the joint optimisation of the clustering and an autoencoder work properly, as the chosen clustering algorithm HDBSCAN introduces multiple challenges as it uses densities to estimate the numbers of clusters.

We showed that the proposed architecture using an undercomplete autoencoder, followed by clustering on the learned embeddings, was capable of performing better than all binning tools included in our comparison on the CAMI high dataset. However, using the same autoencoder and weighting of features did not produce good results when run on the more complex dataset CAMI Airways. By tweaking parameters of AutoBinner on the CAMI high dataset and then using the CAMI airways dataset for validation ensured that the reported results did not give an unfair impression of AutoBinner. We did however show, that by simply adjusting the weighting scheme of the abundance and composition features, enabled AutoBinner to perform well on the CAMI Medium dataset. Minor tweaking of the HDBSCAN settings did also show promising results on the CAMI Airways dataset, emphasizing the potential of the proposed method, AutoBinner.

Appendix A

A.1 AMBER Metrics

The metrics are split into two categories; Metrics per bin, i.e. how did the binner perform on individual bins, and metrics per sample, where the binner is evaluated based on its overall performance on all bins.

Metrics per bin concerns the evaluation of the individual bins, where the focus is to see how well the binner has been able to find the correct bins. This means that for a given bin, it is only the bps/contigs that have been put in that bin, that is going to be part of the evaluation of that bin.

For the metrics per bin, the following metrics are used:

- **Predicted bin size in bps and sequences**
  The predicted size of the bin is the sum of bps/contigs that the binner has placed in a given bin. The true sizes of the bins are also included, such it can be used to evaluate how close the binner is to create bins of the correct sizes.
- **True positives**
  The amount of bps/contigs that have been correctly assigned to a given bin.
- **Purity**
  Purity, which in other contexts is also known as precision or specificity, is a measurement that denotes the fraction of correctly assigned bps/contigs of all the bps/contigs that were assigned a given bin. The purity is calculated as follows

  $$Purity\ p = \frac{TP}{TP + FP}$$

  where TP are the true positives, and FP are the false positives, i.e. the bps/contigs that have been put into the bin, but does not belong there.
- **Completeness**
  Also known as recall or sensitivity, completeness measures how many of the contigs that should be in a bin, have actually been placed in that bin. Completeness is calculated as follows

  $$Completeness\ r = \frac{TP}{TP + FN}$$

  where again TP is the true positives and FN is the false negatives, i.e. the bps/contigs that should have been in a bin, but was put elsewhere.

Metrics per sample is where AMBER looks at all the bins as a whole, in order to get an overview of how well the binner has performed. Here the bins are weighted based on the size of the bins, such that bins with a large volume of bps/contigs have a higher influence on the metrics than ones with fewer bps/contigs.

For the metrics per sample, the following metrics are used.

- **Purity**
  Purity per sample is the average purity over all the bins predicted by the binner, which is calculated as follows
  $$\hat{p} = \frac{1}{n_p} \sum_{x \in X} p_x$$
  where n_p is the number of predicted bins, p is indexed by x where X is the set of predicted bins.
- **Contamination**
  Contamination per sample is again the opposite of purity and is calculated as follows
  $$\hat{c} = 1 - \hat{p}$$
- **Accuracy**
  The accuracy of the bins is measured similar to purity but over the entire dataset, including the bps/contigs that were not included in any bins. The accuracy is calculated as follows
  $$Accuracy\ a = \frac{\sum_{x \in X} TP}{U + \sum_{x \in X} TP + FP}$$
  where U is the set of contigs that have not been put in a bin by the binner.
- **Completeness**
  The average completeness of the sample is calculated not just by the predicted bins, as the case with purity, but here all the bins from ground truth are used. It is defined as follows
  $$\hat{r} = \frac{1}{n_r \sum_{x \in X} r_x}$$
  where n_r is the number of ground truth bins, and X here is the set of all the contigs.
- **Adjusted Rand Index**

  The adjusted Rand index (ARI) is used in AMBER as a metric to express the similarity between the predicted bins made by the binner, and the actual bins from the ground truth. This is done by comparing each bin made by the binner with each bin from the ground truth. The normal Rand index produces a result between 0 and 1, where 0 is no similarity and 1 is complete similarity. However, the Rand index is susceptible to change, which is why AMBER uses the ARI instead. To avoid being susceptible to change, ARI incorporates the expected value of the Rand index, as well as a normalisation factor, which can be seen in the following equation, where $E[RI]$ is the expected value of the Rand index and Norm is the normalisation.

  $$ARI = \frac{RI - \mathbb{E}[RI]}{Norm - E[RI]} \tag{A.1}$$

  Compared to the normal Rand index, the ARI can potentially output a similarity score that is below 0, but still with a max similarity of 1. The ARI computes the similarity between the bins, by creating a contingency table between the bps/contigs in the predicted bins and the bps/contigs in its underlying genome. In Equation A.2

is the ARI definition for AMBER, where m is the contingency table between a given bin x and its underlying genome y, n is the number of elements in m. $m_{x,.}$ and $m_{.,y}$ are the marginal totals of x and y in m, i.e. $m_{x,.} = \sum_y m_{x,y}$ and $m_{.,y} = \sum_x m_{x,y}$.

$$ARI_{AMBER} = \frac{\sum_{x,y}\binom{m_{x,y}}{2} - \frac{\sum_x \binom{m_{x,.}}{2}\sum_y \binom{m_{.,y}}{2}}{\binom{n}{2}}}{\frac{1}{2}\left(\sum_x \binom{m_{x,.}}{2} + \sum_y \binom{m_{.,y}}{2}\right) - \frac{\sum_x \binom{m_{x,.}}{2}\sum_y \binom{m_{.,y}}{2}}{\binom{n}{2}}} \tag{A.2}$$

The equation in Equation A.2 is a bit hard to grasp, so lets split it up a bit to see how it fits with Equation A.1. First, we have the Rand index in the upper left corner

$$RI = \sum_{x,y}\binom{m_{x,y}}{2}$$

Next, we have the normalisation in the bottom left corner

$$Norm = \frac{1}{2}\left(\sum_x \binom{m_{x,.}}{2} + \sum_y \binom{m_{.,y}}{2}\right)$$

Finally, in the upper and lower right corner, we have the expected value of the Rand index between the predicted bin and its underlying genome

$$\mathbb{E}[RI] = \frac{\sum_x \binom{m_{x,.}}{2}\sum_y \binom{m_{.,y}}{2}}{\binom{n}{2}}$$

- **Percentage of binned bps and sequences**
  As the name suggests, this metric measures how large a percentage of the bps and sequences (contigs) that have been binned. Note that this metric does not take into account whether the assignments are correct or not.
- **Number of genomes recovered within levels of completeness and contamination**
  This metric is a table that shows how many genomes, i.e. bins, that the binner has found within different thresholds. For each binner, the thresholds are set at $< 5\%$ and $< 10\%$ contamination and $> 50\%$, $> 70\%$ and $> 90\%$ completeness. The metric gives an overview of how many correct bins the binner has made, as well as the quality of the bins.

A.2 Preparing the data

Before being able to run our binner, we first need to prepare the data. This is done by first mapping the contigs to the reads that they were assembled from. To do the mapping

we use the tool Minimap2 by Li [2018], which produces the BAM files that we need for the next step of preprocessing the data. The next step is to sort the BAM files, which we do by using the sort function from the tool Samtools, by Li et al. [2009]. Both Minimap2 and Samtools were run with their default parameters, we only changed the parameters regarding the system it was run on. Now all the files are ready, and AutoBinner can be run.

A.3 Contractive Autoencoder

A contractive autoencoder (CAE) [Rifai et al., 2011] is used with *overcomplete* autoencoders, i.e. the opposite of the undercomplete autoencoder, and has more neurons in the latent layer than input features. An overcomplete autoencoder has the problem that it can learn the trivial solution, which is when the input is mapped directly to the hidden layer such that the hidden layer simply outputs the input features. A CAE is a way to get around the problem.

A CAE uses a term added to the loss function, like $L1$ and $L2$ regularisation, that explicitly penalizes the trivial solution. Different from $L1$ and $L2$ is that the term disregards the weight of the model and instead penalizes the model based on the partial derivatives of the input features. The term added to the loss function is the squared Frobenius norm of the Jacobian matrix $J_f(x)$ as shown in Equation A.3. The Jacobian matrix $J_f(x)$ is thus a matrix with dimensions $i \times j$ where i is the number of input feature dimensions and j is the number of neurons in the latent layer. The matrix holds all the partial derivatives of all extracted features (the output of the latent layer) with respect to the input feature dimensions.

$$||J_f(x)||_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2 \tag{A.3}$$

Adding this term to the loss can be interpreted as not wanting the encoder to extract any information, as the AE is punished when the input influences the encoding. Minimising this loss term alone would create an AE that learns absolutely nothing. However, when the reconstruction loss is also part of the AEs loss function, an interesting negative synergy between the two loss terms is created. The AE learns to only reflect the variations of the training data in the latent layer. Ideally, the AE also learns to be invariant to other kinds of variation that is not present in the training data. In this way, it would make the encoding of x identically to the encoding of $\hat{x}$ that has added noise.

Bibliography

Aggarwal, **2018**. Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018.

Albertsen et al., **2013**. Mads Albertsen, Philip Hugenholtz, Adam Skarshewski, Kåre L Nielsen and Gene W Tyson. *Genome sequences of rare, uncultured bacteria obtained by differential coverage binning of multiple metagenomes*. `https://www.nature.com/articles/nbt.2579`, 2013. cf. 8-06-2020.

Alneberg et al., **2013**. Johannes Alneberg, Brynjar Smari Bjarnason, Ino de Bruijn, Melanie Schirmer, Joshua Quick, Umer Z. Ijaz, Nicholas J. Loman, Anders F. Andersson and Christopher Quince. *CONCOCT: Clustering cONtigs on COverage and ComposiTion*. `https://arxiv.org/abs/1312.4038`, 2013. cf. 22-05-2020.

Bowers et al., **2017**. Robert M Bowers, Nikos C Kyrpides, Ramunas Stepanauskas, Miranda Harmon-Smith, Devin Doud, T B K Reddy, Frederik Schulz, Jessica Jarett, Adam R Rivers, Emiley A Eloe-Fadrosh, Susannah G Tringe, Natalia N Ivanova, Alex Copeland, Alicia Clum, Eric D Becraft, Rex R Malmstrom, Bruce Birren, Mircea Podar, Peer Bork, George M Weinstock, George M Garrity, Jeremy A Dodsworth, Shibu Yooseph, Granger Sutton, Frank O Glöckner, Jack A Gilbert, William C Nelson, Steven J Hallam, Sean P Jungbluth, Thijs J G Ettema, Scott Tighe, Konstantinos T Konstantinidis, Wen-Tso Liu, Brett J Baker, Thomas Rattei, Jonathan A Eisen, Brian Hedlund, Katherine D McMahon, Noah Fierer, Rob Knight, Rob Finn, Guy Cochrane, Ilene Karsch-Mizrachi, Gene W Tyson, Christian Rinke, The Genome Standards Consortium, Alla Lapidus, Folker Meyer, Pelin Yilmaz, Donovan H Parks, A Murat Eren, Lynn Schriml, Jillian F Banfield, Philip Hugenholtz and Tanja Woyke. *Minimum information about a single amplified genome (MISAG) and a metagenome-assembled genome (MIMAG) of bacteria and archaea*. `https://www.nature.com/articles/nbt.3893`, 2017. cf. 08-06-2020.

Brownlee, **2017**. Jason Brownlee. *Better Deep Learning - Train Faster, Reduce Overfitting, and Make Better Predictions*. 2017. `https://machinelearningmastery.com/better-deep-learning/`.

Chen et al., **2017**. Dongdong Chen, Jiancheng Lv and Zhang Yi. *Unsupervised Multi-Manifold Clustering by Learning Deep Representation*. `https://aaai.org/ocs/index.php/WS/AAAIW17/paper/view/15099/14689`, 2017. cf. 1-06-2020.

Djork-Arne Clevert, **2016**. Thomas Unterthiner Sepp Hochreiter Djork-Arne Clevert. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUS)*. `https://arxiv.org/pdf/1511.07289.pdf`, 2016. cf. 26-05-2020.

Goodfellow et al., **2016**. Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Géron, **2019**. Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow SECOND EDITION*. 2019.

Kang et al., **2019**. Dongwan D. Kang, Feng Li, Edward Kirton, Ashleigh Thomas, Rob Egan, Hong An and Zhong Wang. *MetaBAT 2: an adaptive binning algorithm for robust and efficient genome reconstruction from metagenome assemblies.* https://peerj.com/articles/7359/, 2019. cf. 22-05-2020.

Kingma and Ba, **2015**. Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* https://arxiv.org/pdf/1412.6980.pdf, 2015. cf. 26-05-2020.

Koslicki et al., **2017**. David Koslicki, Ivan Gregor, Stefan Janssen and Stephan Majda. *Critical Assessment og Metagenome Interpretation - A benchmark of metagenomic software.* https://www.researchgate.net/publication/320177427_Critical_Assessment_of_Metagenome_Interpretation_-_A_benchmark_of_metagenomics_software, 2017. cf. 23-05-2020.

Li et al., **2020a**. Fei-Fei Li, Ranjay Krishna and Danfei Xu. *Neural Networks Part 1: Setting up the architecture.* https://cs231n.github.io/neural-networks-1/, 2020. cf. 19-05-2020.

Li et al., **2020b**. Fei-Fei Li, Ranjay Krishna and Danfei Xu. *Backpropagation, Intuitions.* https://cs231n.github.io/optimization-2/, 2020. cf. 19-05-2020.

Li et al., **2020c**. Fei-Fei Li, Ranjay Krishna and Danfei Xu. *Neural Networks Part 2: Setting up the Data and the Loss.* https://cs231n.github.io/neural-networks-2/, 2020. cf. 19-05-2020.

Li et al., **2020d**. Fei-Fei Li, Ranjay Krishna and Danfei Xu. *Linear classification: Support Vector Machine, Softmax.* https://cs231n.github.io/linear-classify/, 2020. cf. 20-05-2020.

Li et al., **2020e**. Fei-Fei Li, Ranjay Krishna and Danfei Xu. *Optimization: Stochastic Gradient Descent.* https://cs231n.github.io/optimization-1/, 2020. cf. 19-05-2020.

Li, **2018**. Heng Li. *Minimap2: pairwise alignment for nucleotide sequences.* https://academic.oup.com/bioinformatics/article/34/18/3094/4994778, 2018. cf. 09-06-2020.

Li et al., **2009**. Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis and Richard Durbin. *The Sequence Alignment/Map Format and SAMtools.* https://pubmed.ncbi.nlm.nih.gov/19505943/, 2009. cf. 09-06-2020.

Lin and Liao, **2016**. Hsin-Hung Lin and Chieh Liao. *Accurate binning of metagenomic contigs via automated clustering sequences using information of genomic signatures and marker genes.* https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4828714/, 2016. cf. 10-06-2020.

Linnebjerg et al., 2019. Simon Nagel Linnebjerg, Mathias Lundhede Hansen and Lasse Erritsø Vive. *A study on metagenomic binning and the underlying clustering approaches*. https://projekter.aau.dk/projekter/da/studentthesis/a-study-on-metagenomic-binning-and-the-underlying-clustering-approaches(0bd47c0c-7a92-418e-.html, 2019. cf. 14-05-2019 - Requires AAU login.

McInnes et al., mar 2017. Leland McInnes, John Healy and Steve Astels. *hdbscan: Hierarchical density based clustering*. The Journal of Open Source Software, 2(11), 2017. doi: 10.21105/joss.00205. URL https://doi.org/10.21105%2Fjoss.00205.

McInnes et al., 2018. Leland McInnes, Victor Shih, Robert Watkins, Nathaniel Saul, Mark Dimitsas and Lukas Grossberger. *How HDBSCAN Works*. https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html, 2018. cf. 10-06-2020.

Meyer et al., 2018. Fernando Meyer, Peter Hofmann, Peter Belmann, Ruben Garrido-Oter, Adrian Fritz, Alexander Sczyrba and Alice C. McHardy. *AMBER: Assessment of Metagenome BinnERs*. https://academic.oup.com/gigascience/article/7/6/giy069/5034950, 2018. cf. 19-05-2020.

Min et al., 2018. Erxue Min, Xifeng Guo, Qiang Liu, Gen Zhang, Jianjing Cui and Jun Long. *A Survey of Clustering With Deep Learning: From the Perspective of Network Architecture*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8412085, 2018. cf. 15-04-2020.

National Research Council; Division on Earth and Life Studies, 2007. National Research Council; Division on Earth and Life Studies. *The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet*. https://www.nap.edu/read/11902/chapter/3#31, 2007. cf. 09-06-2020.

Ng, 2020. Andrew Ng. *Normalizing inputs - Practical aspects of Deep Learning*. https://www.coursera.org/lecture/deep-neural-network/normalizing-inputs-lXv6U?fbclid=IwAR1evCxLcOCVCLn1CnOGnFBF_MSXY-jCNp0l7VckduynBjLbbEqaxdqnlqg, 2020. cf. 8-06-2020.

Ng, 2011. Andrew Ng. *CS294A Lecture notes*. https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf, 2011. cf. 20-05-2020.

Nissen et al., 2018. Jakob Nybo Nissen, Casper Kaae Sønderby, Jose Juan Almagro Armenteros, Christopher Heje Grønbech, Henrik Bjørn Nielsen, Thomas Nordahl Petersen, Ole Winther and Simon Rasmussen. *Binning microbial genomes using deep learning*. https://www.biorxiv.org/content/10.1101/490078v2.article-info, 2018. cf. 15-04-2020.

Parks et al., 2014. DH Parks, M Imelfort, CT Skennerton, P Hugenholtz and GW Tyson. *Assessing the quality of microbial genomes recovered from isolates, single*

cells, and metagenomes. https://genome.cshlp.org/content/25/7/1043, 2014. cf. 05-06-2020.

Rifai et al., **2011**. Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot and Yoshua Bengio. *Contractive Auto-Encoders: Explicit Invariance During Feature Extraction*. https://icml.cc/2011/papers/455_icmlpaper.pdf, 2011. cf. 22-05-2020.

Xie et al., **2016**. Junyuan Xie, Ross Girshick and Ali Farhadi. *Unsupervised Deep Embedding for Clustering Analysis*.
http://proceedings.mlr.press/v48/xieb16.pdf, 2016. cf. 29-05-2020.