B.Sc.Eng. Thesis
Bachelor of Science in Engineering

Convolutional neural networks for unsupervised learning of patterns in DNA sequences

Theis Ferré Hjortkjær

The Technical University of Denmark 2019

Contents

Preface

This report is the product of a 15 ECTS Bachelor thesis for the completion of my BSc. Eng. Biotechnology at The Technical University of Denmark (DTU). The project was conducted from the 1st of february to the 5th of June 2019 at the facilities of Novo Nordisk Center for Protein Research (CPR) in cooperation with the Rasmussen group that focuses on Human Proteome Variation. The external supervisor from CPR was group leader Simon Rasmussen. The internal supervisors from The Technical University of Denmark for the project was Henrik Nielsen and Jose Juan Almagro Armenteros.

I would like to thank Henrik and Jose for saying yes to being my internal supervisors from DTU and also helping me getting started with the project. I would also like to thank Jakob Nybo Nissen from Rasmussen group for his great academic support and always helping me whenever needed. A special thanks to Simon for having a project that I found interesting and also introducing me to his great research group.

Theis Ferré Hjortkjær
S163700

Abstract

Next-Generation-Sequencing has allowed scientist to develop a high-throughput method for sequencing DNA with low cost and high precision. This has led to advances in bioinformatics and especially metagenomics. In metagenomics the goal is to identify and reconstruct microbial species directly from environmental samples. This allows scientist to discover new organisms that could potentially contain new biological pathways with great interest for many industries.

In this report, the possibiliies of combining deep learning techniques and metagenomics was discovered, by creating a Convolutional Variational AutoEncoder that was trained directly on contigs to cluster sequencing data based on taxonomy. It was found that using Convolutional layers in the Variational AutoEncoder had a detrimental effect on the results, because of the requirement that each input sequence had to be the same length, which forced the contigs to be fragmented. This showed to weaken the signal in the data to such a degree that the model was not able to create meaningful clusters.

Chapter 1

Introduction

Metagenomics is currently a field of interest for many bioinformaticians with challenges to overcome, and great potential for discovering previously unknown organisms that can help us understand microbial communities and their interactions. As sequencing technology becomes cheaper and more efficient the amount of metagenomics data is increasing rapidly (Drmanac 2011). This puts a high demand on developing fast and precise algorithms for analyzing and interpreting the data. The data generated from metagenomic seqencing is a massive amount of DNA reads. These reads can be combined into longer fragments called 'contigs' based on overlaps in reads.

In this project, alternative approaches for binning/clustering contigs from metagenomics sequencing data, will be discovered. Current approaches rely, among others, on Tetra Nucleotide Frequency (TNF) information from the contig sequence composition. This method assumes that organisms have a specific pattern in the way their DNA is coded and that every organism has its own unique TNF composition.

The approach to binning/clustering, in this project, will be to create a neural network that will identify DNA patterns directly from the contig sequences, without using the contig TNF profiles. Instead, large contigs will be split into smaller fragments of equal size. Each contig will be represented by its contig-fragments which the model will analyze and compile into one result which will correspond to that contig. The network architecture for doing this will be a Convolutional Variational AutoEncoder (CVAE) that takes the raw contig-fragments as input and encodes a latent representation for all of these. The goal for the model is to create a latent representation where contig-fragments together represent the contig that they belong to. This means that every contig will have its own latent representation. With these latent representations there will be created clusters that should make up parts

of genomes such that the clusters, with high confidence, can show what organisms is in the data. The Cami Airways (https://data.cami-challenge.org/participate) dataset will be used to train and evaluate the model. The model will then be benchmarked against VAMB (Nissen et al. 2018) using a clustering and binning algorithm developed in this study. VAMB takes a similar approach, in using a Variational AutoEncoder to unsupervised cluster contigs. The main difference is, that VAMB uses pre-calculated TNF compositions as an input for the model. This also makes VAMB a good benchmark, since it will show how the two different approaches compare in learning DNA patterns, with the same underlying data. The code used to create the model and analyze/pre-process the data can be found in the following GitHub Repository: (**https://github.com/TheisFerre/CVAE**)

Chapter 2

Theory

2.1 Variational Autoencoder

To understand a Variational AutoEncoder, one must understand the basic principles of a simple neural network. Neural networks and deep learning are a subset of machine learning. Machine learning is the science of getting a computer to learn how to perform a task. What makes neural networks unique, is that they can be as complex as the programmer makes them. Therefore they can also learn complex functions that simpler "out of the box" machine learning algorithms or humans cannot.

A neural network is a computational graph consisting of several layers. The first layer (input layer) receives an input that is computed through the layers of the network. In each layer there is a number of nodes (neurons) that connects to the neurons in the next layer through weights called θ (see **Figure 2.1**). Each neuron in the layer l is connected to every neuron in layer $l+1$. The input for the neuron in the next layer is the sum of all the weights multiplied with their signal from the corresponding neuron. This input is then sent through an activation function that must "activate" the neuron if a certain criterion from the activation function is met. Activation functions that are commonly used are ReLu that computes max(0, x) and sigmoid that computes $\frac{1}{1+e^{-x}}$. This whole process is also called the feedforward algorithm as it feeds the input through each layer and finally computes an output.

All the layers between the input and output layer is called hidden layers, as the purpose of these are solely to enable the network to learn a function that computes an output that will reduce a loss function (Sazli 2006).

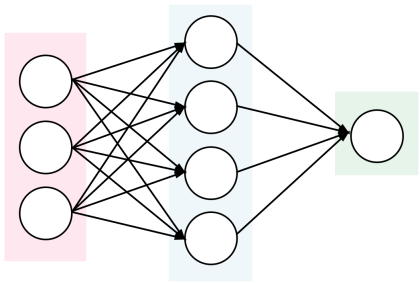

Figure 2.1: A standard feed forward neural network. The three left-most neurons are the input layer, the four neurons in the middle is a hidden layer and the right-most neuron is the output layer. In this case, the output layer only consist of one neuron. Every neuron in one layer is connected to every neuron in the next layer with some weights θ

This loss function is the optimization problem of the network. The objective is to minimize the loss function through optimization of the weights. Therefore, it is crucial to define a loss function that properly encompasses the objective of the network such that the weights can be properly optimized, allowing the network to learn.

To optimize the weights in neural networks, an algorithm called backpropagation is used, that utilizes stochastic gradient descent(Robbins and Monro 1951). Unlike the feed forward algorithm, backpropagation starts at the output layer where it computes the error at each layer. This error originates from the loss function and by using the derivatives of the activation functions and the chain rule, the weights at each layer can be tweaked with the scope of minimizing that layer's contribution to the overall error (loss function) in the network (LeCun 1988).

Suppose, there is over 1000 weights (which is typically a small amount) in the network that has the be tweaked, so that the loss function for every input is minimized. This is a non-convex problem because of the high number of parameters (weights) in the network and the fact that the network must generalize to several different inputs makes it a complicated task and it is easy for the network to get stuck at local minimums in the optimization (Swirszcz, Czarnecki, and Pascanu 2016).

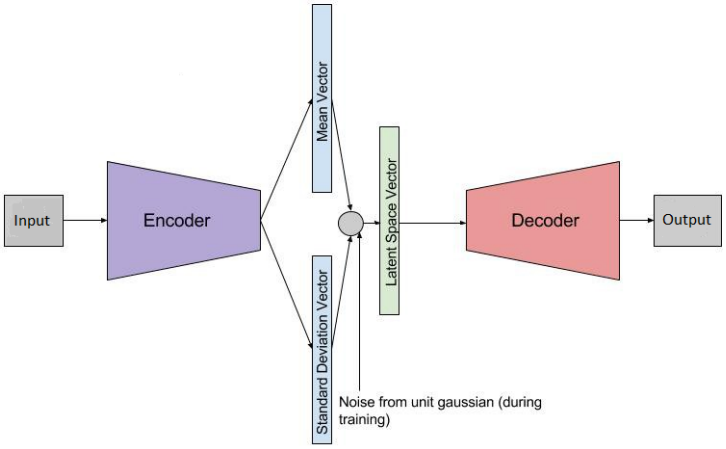

Figure 2.2: The structure of a Variational AutoEncoder. The Variational AutoEncoder receives an input that is send to the Encoder. Here it computes the mean and std. vector that forms the latent space. The latent space is then decoded by the Decoder that finally computes an output. Note that the Decoder is an mirror image of the Encoder such that the input and output has the same shape

In this project a Variational AutoEncoder (VAE) is used (see **Figure 2.2**). A VAE is a special kind of neural network and it has a very similar structure to an autoencoder (Rumelhart, Hinton, and Williams 1986). A standard autoencoder takes a high dimensional input and sends it through some encoding layers where the dimension is compressed. This compressed layer is called the latent space. The network also has a decoder that can take the compressed latent space and decode it into a output with the same dimension as the input. This means that the architechture of the decoder is a mirror image of the encoder. The goal is that the output must be similar to the input through optimization of the weights θ. This can be used for several things, such as denoising and compression. VAE's has the same structure as a standard AutoEncoder, but they are used for different things.

VAE's are categorized as unsupervised generative models that learns to model a distribution of datapoints X, where X is samples from a high dimensional dataset. Images are used very often for this process, but it can be used for anything where there is an interest in modelling the distribution of the data. The special thing about VAE's is that they enable sampling of a vector z in the latent space. It works by making the encoder compute two vectors with the same dimension as z. One vector represents the mean μ and the other one represents the standard deviation σ in a gaussian distribution. These vectors are combined with a standard normal *N(0, I)*,

to form the latent space z, where I is the identity matrix with the same dimension as σ and μ, see equation 2.1. (Kingma and Welling 2014).

$$z = N(0,I) \odot \sigma + \mu \tag{2.1}$$

This allows for sampling of vectors with the same dimension as z where the elements are gaussian distributed. The decoder maps the sampled vector to a function f that is parameterized by θ such that $f(z;\theta)$ will produce an output similar to the data that has been presented to the model (Doersch 2016) see **Figure 2.3**.

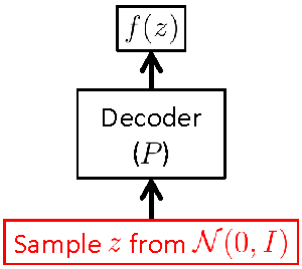

Figure 2.3: The decoder can take gaussian distributed samples from z and map it through a function $f(z)$ that is able to produce outputs similar to the learned distribution from the data the network has been trained with

By making the encoder compute two vectors that represent the parameters of a gaussian distribution, continuity is maintained in the network. This is a crucial part of making backpropagation work, such that it can be used to learn a optimal set of weights θ. This is called the reparameterization trick. If the parameters of the model are optimized, gaussian distributed samples of z will map to a distribution of generated data that is similar to the data presented to the network. Although these distributions will be similar, they do not contain the same data points. This is the generative part of the model where it creates "new" data, that with a high probability, comes from the same distribution as the original data.

Since this reparameterization trick is introduced, the Kullback-Leibler Divergence (KL-Divergence) needs to be added to the loss function. The KL-Divergence term represents the difference between two distributions. The KL-Divergence loss is calculated based on σ and μ that form the latent space. Intuitively, the KL-Divergence keeps the latent space Gaussian distributed. The formula for this is as follows:

$$KL\,Divergence = -\frac{1}{2}\sum_{i=1}^{N}(1 + log(\sigma_i) - \mu_i^2 - \sigma_i) \tag{2.2}$$

Where N is the dimension of the vectors σ and μ.
This means that there is two things to optimize in the network through loss functions. First, the VAE should encode a datapoint and then decode it such that the output from the decoder is similar to the input datapoint. This is called the reconstruction loss. second, it should be possible to sample gaussian distributed vectors in the latent space, such that the decoder is able to produce an output similar to the data distribution it has been trained on. This is the KL-Divergence loss that forces the network to keep the latent space represented as a gaussian distribution. By having two loss functions to minimize, more complexity is introduced in the training of the model since there will be a trade-off between the two. To control how the model trains, weights will be added to each loss function to weight down the loss function that has the least importance for the specific purpose of the model. Thereby, the network can be manipulated to behave in a way that suits the purpose for the project.

Since this project is not about generating data, but rather clustering of the data, the KL-Divergence will be weighted down. Intuitively, this means that the network is allowed to separate the datapoints further away from the center of a gaussian distribution by making the network penalize KL-Divergence more loosely. Doing this, will affect how the datapoints in the latent space is represented and hopefully it will give a more disentangled latent representation, where similar datapoints are encoded together in clusters. These clusters will represent how the VAE encodes the data, such that it is easier for it to reconstruct and thereby minimizing the reconstruction loss. This is also why the weight for the reconstruction loss will be higher than for the KL-Divergence.

2.2 Convolutions for pattern recognition

In the section above, some of the theory behind standard feed forward neural networks and Variational AutoEncoders was explained. The standard feed forward neural network is the most commonly used network type and is great for input data that is structured. This makes the network easy to use but also not very versatile. To overcome the problems with using unstructured data as an input for neural networks there has been developed other network types.

Convolutional Neural Networks is a network type that handles unstructured high dimensional data very well. The most obvious example is images where 2d-convolutions are used. Images are represented as numerical pixel values in a matrix, where each value is one pixel. Furthermore, color images have three channels, each representing

an intensity of the colors red, green and blue (RGB-values). This means that an image can be represented as three matrices with the same dimension. If a standard feed forward neural network was used for the analysis of this, the image would have to be flattened which would ruin the pattern in the image. Therefore, this is not the optimal way of modelling this type of data. Instead, convolutional neural networks are used (LeCun et al. 1989). Convolutions consist of filters which are sliding windows with a determined size where each element in the filter is a parameter that can be optimized by the neural network. The filter starts at the upper left corner of the input and then simply slides through the rows and columns with a determined parameter which is the stride. As a default the stride is 1. At every stride the filter computes the sum of the elementwise product between the filter and the input (see **Figure 2.4**). When the filter has visited every location of the input, an output matrix has been computed. Additionally, one can add padding to the input such that the corners and edges of the input are visited as many times as the elements in the middle. Padding is represented as zeros around the original input. The size of the output matrix can be calculated in the following way, where it is assumed that the width/height is the same:

$$O = \frac{W - K + 2P}{S} + 1 \tag{2.3}$$

O is the output size, W is the input size, K is the filter size, P is the padding and S is the stride.

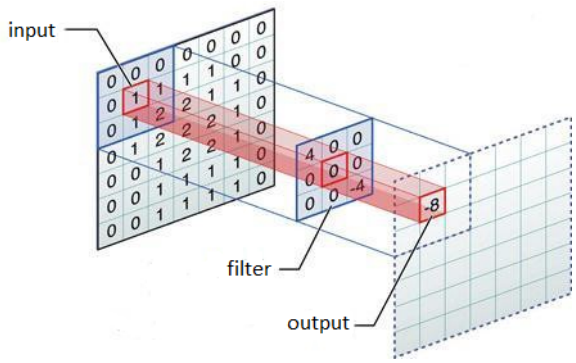

Figure 2.4: A convolutional filter being applied to an input. It computes the sum of the elementwise multiplication between the input and the filter. In this case it results in an output of -8. This multiplication is being done at every possible location in the input

When the filter size, stride and padding has been chosen it can also be specified how many filters to be used. Each filter will then represent a channel in the output such that the output actually consist of many layers of matrices stacked on top of each

other. This enables the filters to optimize their weights for recognizing different patterns. As an example, when working with images of faces, one filter could be optimized to detect the eyes while another filter could be optimized to detect the nose. In modern computer vision tasks it is not unusual to see huge convolutional neural networks to classify images. An example could be Googles Inception net with 22 layers (Szegedy et al. 2015)

In the VAE, the Encoder will consist of convolutional layers to detect patterns in DNA. The convolutional layers will apply filters with strides that scale the input size down such that the latent space will be a compressed representation of the input data. To scale the latent space back to the input size, the Decoder will consist of transposed convolutional layers (Dumoulin and Visin 2018) that simply reverse the Encoder (see **figure 2.5**). For each convolutional layer in the Encoder, there will exist a corresponding transposed convolutional layer in the Decoder that takes the same parameters. This means that when constructing the VAE, the same parameters for the Encoder and Decoder can be used. Since the Encoder and Decoder consist of these convolutional layers it will be referred to as a Convolutional Variational AutoEncoder (CVAE).

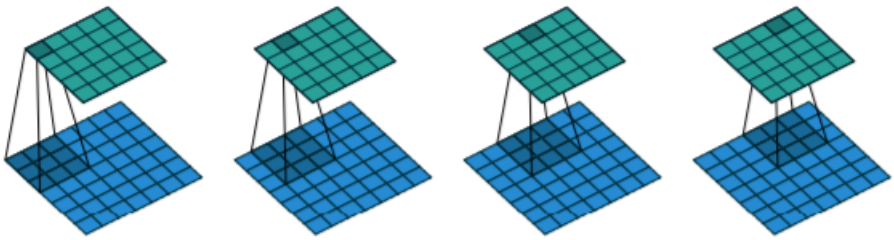

Figure 2.5: Transposed convolution. A filter is applied upon the green matrix which is the input. Here the filter upscales the green matrix into the blue matrix by applying the filter to every element of the input matrix. This results in the blue matrix being bigger than the green input matrix

2.3 Binning of Contigs

The focus of the project will be on binning/clustering the contigs based on their taxonomy (see **Figure 2.6**). This means that the CVAE should be trained to identify which contigs are closely related. In theory, the CVAE will encode closely related inputs similarly. Therefore, the latent space of all the contigs can be clustered and hopefully the bins/clusters contain contigs that are similar. For this to work, it

is assumed that organisms has a preference in how their DNA is structured and that the DNA contains patterns unique to every organism. These patterns is what the convolutional layers in the CVAE is supposed to learn and encode. Using raw sequence data is not the typical approach for binning contigs which is what makes this approach unique to others.

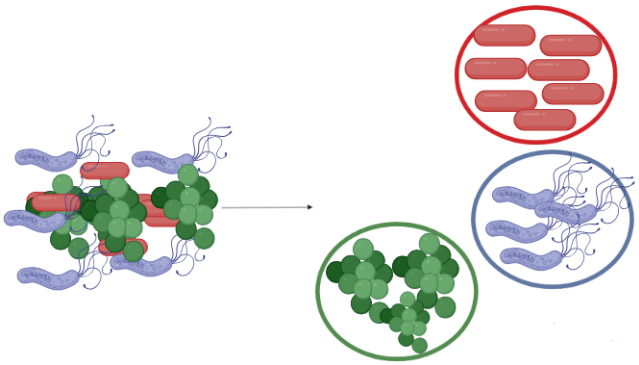

Figure 2.6: Binning of cells based on taxonomy

Typical approaches to binning use Tetranucleotide Frequency (TNF) (Saeed, Tang, and Halgamuge 2011) and abundance data (Wang et al. 2012), sometimes in combination and sometimes not. While these approaches are getting better and better, there is room for improvement and experimentation. The TNF composition of a sequence is generated by taking a sliding window of 4 bases and calculate the frequency of how often each 4-base combination is seen. Since it is unknown whether the DNA string is read from the forward or the backwards DNA strand, the TNF composition is represented in 136 different values. An example of two 4-mers that are equal would be 'ATCC' and 'GGAT'. Using TNF composition data raises some concerns. It assumes that organisms have a specific pattern in the way their DNA is coded and that every organism has its own unique 4-mer composition, such that an algorithm/statistical software is able to separate the contigs based on this unique composition. Furthermore, it assumes that every contig that belongs to a unique microbe will have a similar DNA composition. Keeping these concerns in mind, there has been conducted studies that show that microbes do have a tetranucleotide usage pattern which supports the usage of TNF in binning contigs (Pride et al. 2003). While the CVAE also assumes a pattern in how DNA is coded, it does not assume that it can be captured in a sliding window of 4 bases. Instead, the CVAE

can use convolutional filters of different sizes, where there is no limit to what size of the filter that can be used. This gives a freedom in creation of the CVAE and it also allows for testing of different sizes to find the ones that are optimal.

Chapter 3

Methods

3.1 Data Pre-Processing

The dataset which will be used for the project is the Cami Airways dataset which consists of 187685 contigs. The minimum length of the contigs are 2000 bases and the contig that has the most bases has a length of 6186826, to see how the contig lengths are distributed see **Appendix A1**. Because of these varying sizes, the contigs have been processed such that the CVAE can deal with this variation. To do this, each contig was split into pieces of 2000 bases. Doing this means that no contigs will be removed, and the larger ones will simply be split into fragments. The number of fragments for each contig will be $floor(\frac{length(contig)}{2000})$. This means that a contig with length 5500 will be split into two contigs-fragments representing the lengths 0-2000 and 2000-4000 respectively. The rest of the contig (4000-5500) will be discarded.

To facilitate this, the bioinformatics tool BEDTools (Quinlan and Hall 2010) was used. With BEDTools a window file was created. This file contained the name of each contig and the fragment positions. To then process the contigs into the fragments given by the window file, a python script(process_contigs.py) was created. This script took out the fragments of length equal to 2000 for each contig and then OneHot-Encoded every base in the fragment. To OneHot-Encode simply means to represent the base as a number. The model will not be able to compute on non-numerical data, such as the letter 'A', it needs to get it represented in another way. OneHot-Encoding makes a list with length equal to the number of different classes, which in this case is four. Then it represents each class as '1' in the respective position in the list and '0' everywhere else corresponding to the other classes (see **Figure 3.1**).

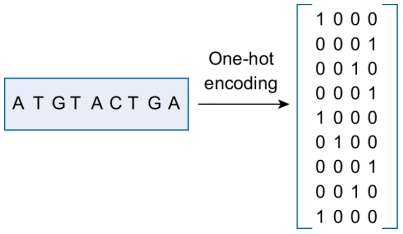

Figure 3.1: OneHot-Encoding of a DNA sequence of length 9

When the file with all the contigs had been processed there was 752780 contig-fragments of length 2000. 300 of these contained basecalls that was not A, G, T or C and was therefore removed from the dataset. Finally, this ended up in a dataset of 752480 contig-fragments. For each contig-fragment there was a OneHot-Encoding with the size [4, 2000], that represented the four bases and the length of the fragment. The dataset was saved with numpy in a .npy file format with the shape [752480,4,2000], the file has a memory usage of 45 Gigabytes. Furthermore, a label file was also created. It contained three columns, representing the organism, contig and fragment position with 752480 lines (one for each contig-fragment). This file will be used later when the results of the unsupervised learning of the model has been done, such that performance of the model can be evaluated.

3.2 Vanilla CVAE

When implementing the model there was one important goal: The implementation had to be versatile, such that it would be easy to test the model with different configurations. This means that the number of layers and hyperparameters should not be hard coded in the model. Instead, the implementation was done such that the network was created directly from the command line. An example of this could be **Figure 3.2** where a Variational AutoEncoder with two convolutional layers in both the Encoder and Decoder is called with channels=[15, 15], filters=[21, 20], strides=[1, 20], padding=[10, 0]. The model was implemented in PyTorch version 0.4.1. (Paszke et al. 2017).

```
[tfehjo@risoe-r07-g003 cnn]$ python main.py --channels 15_15 --kernels 21_20
--strides 1_20 padding 20_0 --save model_1 --cuda
```

Figure 3.2: Creating a Variational AutoEncoder directly from the command line

Since the input data has the shape [752480, 4, 2000], where the first number is the amount of contig-fragments in the dataset, 1d-convolutional layers will be used. This

is done by representing each base in the OneHot-encoding as a channel, meaning that the input has a total of 4 channels each with a length of 2000. 1d-convolutions are similar to 2d-convolutions which are used for images, the only difference is that the input has no height dimension, it only contains a width which is the length of the sequence.

Besides the number of convolutional layers and the parameters of these, the network also consists of other hyperparameters where an optimal setting had to be found. The hyperparameters include learning rate, dropout, KL-Divergence weight(β), latent space dimension and epochs.

The learning rate dictates how fast the weights of the model are optimized. There is a trade-off when choosing the learning rate. If the learning rate is too high, the weights can be changed so much, that it does not actually increase the performance of the model. A low learning rate means that the weights are optimized slowly. This means a longer training time until the model converges. The optimizer using the learning rate will be the Adam Optimizer (Kingma and Ba 2015). The Adam optimizer is a stochastic gradient descent algorithm which combines the advantages of AdaGrad (Duchi and Hazan 2011) and RMSProb (Hinton and Tieleman 2012). It was used with default settings which can be found in the PyTorch documentation.

Dropout (Srivastava et al. 2014) and batch normalization (Ioffe and Szegedy 2015) was added to every layer in the encoding and decoding part of the network. Both Dropout and batch normalization has been shown to increase performance of neural networks in different ways. Batch normalization normalizes layer inputs, which has been shown to speed up the learning of the model without any drawbacks, this makes it an obvious algorithm to introduce to our network layers.

Dropout is a method for regularizing neural networks to prevent overfitting. Dropout randomly 'deactivates' neurons in the neural network (see **Figure 3.3**). This makes the network generalize in a way that makes it robust enough to still make predictions that are correct even though random neurons are switched off and thereby has a regularizing effect. Dropout takes an input for every layer that it is applied to. This input should be a number between 0 and 1, that corresponds to the probability that a node is switched off.

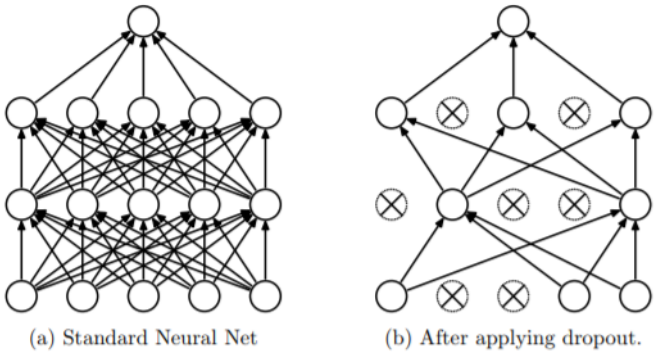

Figure 3.3: Figure from (Srivastava et al. 2014). (a) shows a standard neural network with two hidden layers, where no dropout has been applied. (b) shows the same neural network where dropout has been applied. It displays that some neurons are deactivated and has a regularizing effect

The loss function of the model is the sum of two sub functions: The reconstruction loss and the KL-Divergence, both described in the Theory chapter. When penalizing the model for the reconstruction of the input, the Cross Entropy Loss (CE)(Miranda 2017) was used . The Cross Entropy is calculated by the two formulas below that combines log-softmax (3.1) and Negative Log-Likelihood loss (3.2). The loss is calculated at every One Hot encoded list corresponding to a base in the contig fragment and averaged together.

$$P_i = log(\frac{exp(x_i)}{\sum_{j=1}^{4} exp(x_j)}),\ for\ i = 1..4 \tag{3.1}$$

$$CE = \sum_{i=1}^{4} -log(P_i) \tag{3.2}$$

Furthermore, the KL Divergence was computed as follows:

$$KL\ Divergence = -\frac{1}{2}\sum_{i=1}^{N}(1 + log(\sigma_i) - \mu_i^2 - \sigma_i) \tag{3.3}$$

Where N is the latent space dimension.

The two loss functions was added together to compute the total loss of the model which is what the optimizer uses to apply stochastic gradient descent in the back-propagation. Furthermore, a weighting term β was added to the KL-Divergence. This would serve as a hyperparameter that controls how the model should weight the reconstruction loss vs the KL-Divergence loss. Adding this weighting term has been shown to make Variational AutoEncoders learn a more efficient latent representation which is more disentangled opposed to a Variational AutoEncoder without

it (Higgins et al. 2017). The total loss is therefore:

$$loss = \text{KL-Divergence} \cdot \frac{1}{\beta} + \text{CE} \tag{3.4}$$

β will be set to a value greater than 1. If β is 1 it would have no effect on how the latent space is encoded.

3.3 Gumbel-softmax CVAE

In addition to the Vanilla CVAE there was created another model with a modified Cross Entropy loss function, such that the normal log-softmax was replaced with a Gumbel-softmax. The Gumbel-softmax is similar to the log-softmax, but it has an additional parameter called temperature(τ). This parameter is used to 'exaggerate' the probabilities generated in the normal softmax function. The probabilities are more exaggerated when the temperature is decreased. The intuition behind using the Gumbel-softmax, is that it can be used to make discrete reconstructions of the input while maintaining a differentiable gradient, such that the network can backpropagate based on this (Jang, Gu, and Poole 2016).

When a similar model to the Vanilla CVAE was trained with the Gumbel-softmax, the following annealing schedule for the temperature parameteter was created:

$$\tau = max(0.5, exp(-0.004 * e)) \tag{3.5}$$

Where e is the epoch number.

This annealing schedule would be used to gradually increase the exaggeration of the reconstruction probabilities by decreasing the temperature.

3.4 Model with contig-fragment size 10000

Both the Vanilla CVAE and the Gumbel-softmax CVAE used a minimum contig-fragment size of 2000. To investigate whether this length was too short for the models to detect a signal, it was decided to also create a model where the minimum contig-fragment size was of length 10000. This would be a 5-fold increase in length. To do this, the same pre-processing steps was followed as those mentioned in the **Data Pre-Processing** section, but instead of splitting contigs into pieces of 2000, they were split into pieces of 10000. This meant that the final dataset had the shape [95226, 4, 10000]. Because the minimum contig length in the Cami Airways dataset was 2000, it meant that all the contigs that were smaller than 10000 had to be

discarded. This meant that less data was processed by the model which could have an influence on the performance when benchmarking, because a lot of missing data. Two models was trained with this data, one using the normal Cross Entropy as in the Vanilla CVAE and another one using the Gumbel-Softmax Cross Entropy.

3.5 Binning & Benchmarking

When a model was successfully trained, it had to be evaluated. The evaluation is based upon the latent space where the goal is to have taxonomically related contigs clustered together. Since the latent space is high dimensional it cannot be visualized directly. Instead, a Principal Component Analysis (PCA)(Shlens 2005) can be used, where two axes are captured, that contains the highest variance in a linear combination of every dimension in the high dimensional latent space. This will result in a plot that will show clusters in the latent space if they exist. Since the dataset contains labels for every contig, each contig can be colored according to its taxonomy to see if there exists any clusters of contigs that originate from the same organism.

Doing a PCA only gives visual representation of the latent space and evaluating the performance based on this is not optimal and not objective, since it will be evaluated qualitatively. It can, however, be used as an indication of whether or not the model will cluster the contigs correctly.

To also have a quantitative measure of the clusters in the latent space, a clustering algorithm (see **Appendix A2**) developed in the VAMB project (Nissen et al. 2018) was used. This algorithm uses pearson distance to find clusters in the high dimensional latent space together with a threshold that determines whether or not more contigs form a cluster. The algorithm assigns every contig to a cluster. This means that if there is N contigs, there is a possibility that there will be N unique clusters each containing one contig. If this is the case, the model does not encode the latent space properly and no meaningful bins in the data will be found.

When the contigs were split into clusters, they had to be evaluated whether they were meaningful. To do this a benchmarking script also developed in the VAMB project was used. The script uses the clusters generated by the clustering algorithm, and a reference file. This reference file contains information about each contig. The information consists of the Operational Taxonomic Unit (OTU) identifier, the GenBank identifier and what position in the genome the contig covers. This means, that the reference file contains a set of true bins that our model should replicate.

The benchmarking script uses two metrics to evaluate whether the clusters created by the clustering algorithm can be considered a bin. The two metrics are precision and recall. The calculation of these are described in the VAMB project as follows: *When benchmarking a set of bins against a true set of bins, we matched each true bin with each of the observed bins and for each pair, calculated the precision as the total size of contigs in the observed bin that belonged to the true bin divided by the total size of the observed bin. The recall was defined likewise, but with the size of the true bin in the denominator* (Nissen et al. 2018). The more general formula for precision and recall can be described as in equations 3.6 and 3.7.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{3.6}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3.7}$$

When the different elements of the methods were combined the following general workflow was achieved **Figure 3.4**.

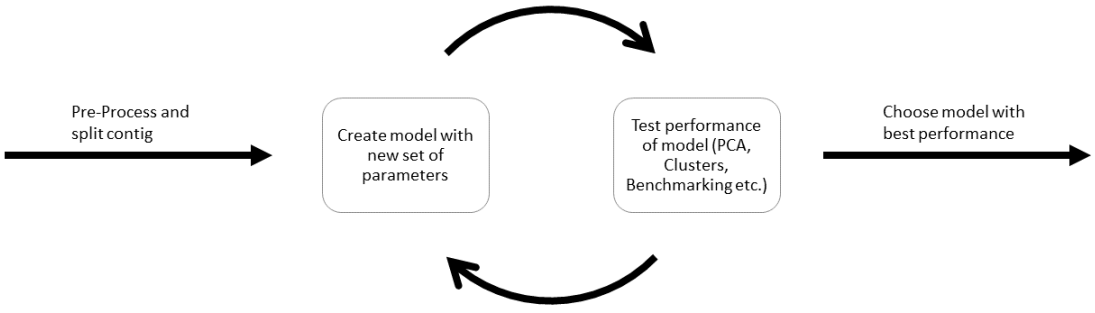

Figure 3.4: Workflow for finding model with best performance

Chapter 4

Results

4.1 Vanilla CVAE

As described in the Methods chapter, it had to be easy to test numerous different models. The reason for this, is that the results potentially could vary a lot by using different hyperparameters. Initially, models that showed no clustering at all in the latent space when using PCA could be discarded. The results in this section are from a model with the following settings:

Epochs	Batch size	Learning Rate	Latent Size	β	Dropout	Channels	Filters	Strides	Padding
250	1028	0.001	40	1000	0.3	[34,34,34,15,15]	[9,15,21,20,10]	[1,1,1,20,10]	[4,7,10,0,0]

This model showed the best performance out of ∼20 tested different models. Since there is 639 different organisms in the dataset a single PCA plot with all these organisms would be a mess. Therefore, the PCA plot contains contigs from only five different organisms (see **Figure 4.1**).

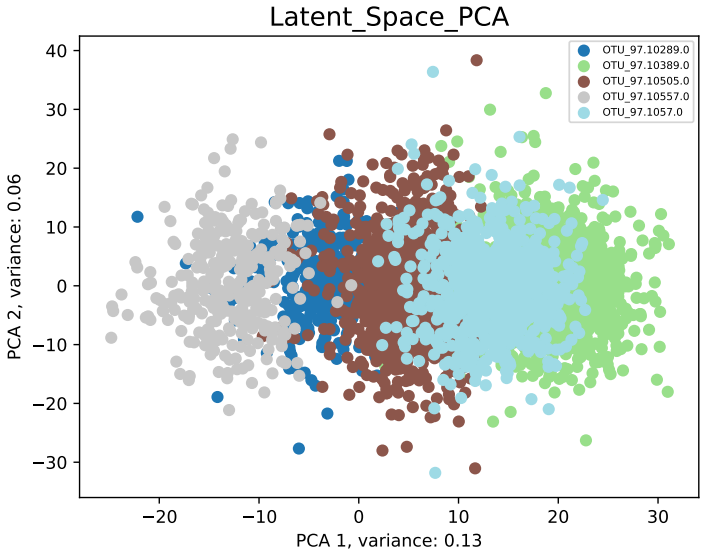

Figure 4.1: PCA plot showing how five different contigs cluster. The explained variance for the first Principal Component is 0.13 and 0.06 for the second

The PCA plot shows that there is some separation based on which organism the contig originates from. However, there is also a clear overlap between some of the clusters. If more than five organisms were displayed, the overlap would become even more obvious.

When training the model it is interesting to see how the loss functions behave. This is displayed in **Figure 4.2**. It is also important to note that the KL-Divergence loss increases. This is the type of behaviour that our model should have. It means that the latent space encoding is spread away from the center of a gaussian distribution. However, even though this behaviour is observed in the KL-Divergence, the Cross Entropy falls a very minor rate which means that the model does not learn how to reconstruct the input that it receives. This could indicate trouble with encoding similar contig-fragments similarly.

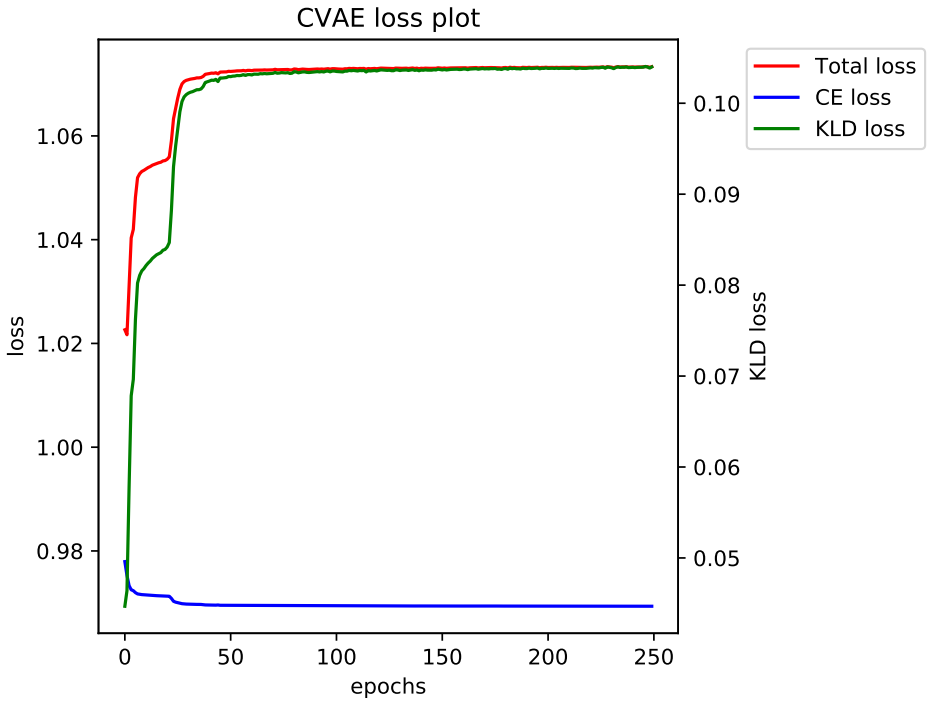

Figure 4.2: Loss plot with two axes. The Total loss and Cross Entropy is plotted on the left axis. The KL-Divergence loss is plotted on the right axis due to it being significant smaller. The reason why it is smaller is because that β is 1000 and the KL-Divergence is multiplied with $\frac{1}{\beta}$

When using the clustering algorithm we find 25 clusters containing 5 or more contigs in the latent space. The largest cluster contains 7263 contigs. When using the binning algorithm on these clusters **Table 4.1** is achieved, which corresponds to the number of bins where each bin has a size $\geq$ 100000 for each precision/recall score:

		Precision					
		0.3	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Recall	**0.3**	5	4	4	3	3	3
	0.5	4	4	4	3	3	3
	0.7	2	2	2	2	2	2
	0.9	2	2	2	2	2	2
	0.95	2	2	2	2	2	2
	0.99	2	2	2	2	2	2

Table 4.1: Number of bins generated by Vanilla CVAE with a minimum bin size of 100000.

In **Table 4.1** only two of the bins contain more than one contig. The two bins that contain more contigs have a precision/recall score of 0.4/0.38 and 0.55/0.89 and they consist of 30 and 8 contigs respectively.

To find out why the number of bins were not higher, it was decided to look at how the contig-fragments for each contig was encoded in every latent space neuron with a boxplot. Optimally, the fragments should create a signal in the latent space by averaging the neurons away from zero.

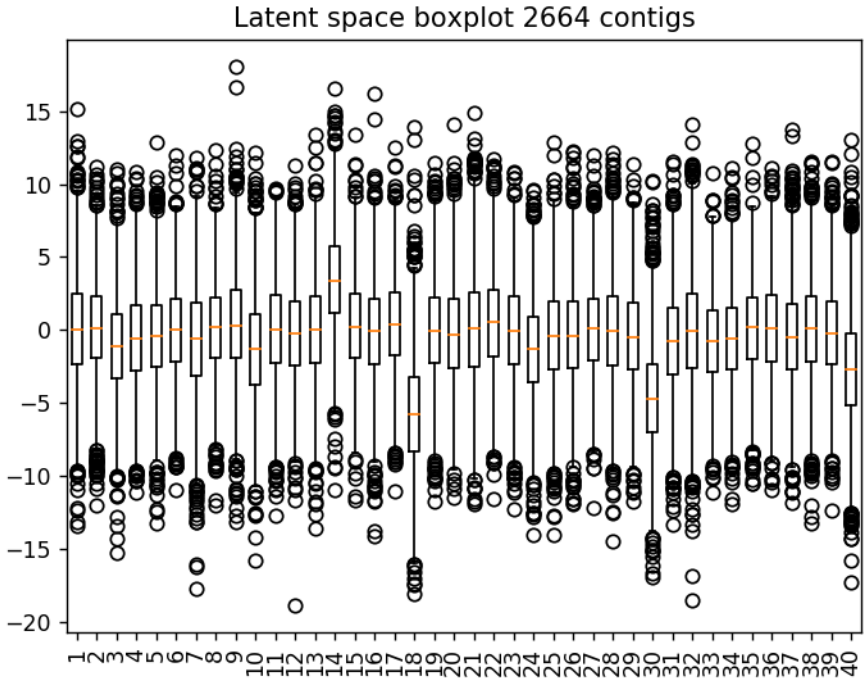

Figure 4.3: Boxplot of a randomly selected contig consisting of 2664 contig fragments. The x-axis is the neurons and the y-axis is where each fragment for that neuron is encoded

By looking at **Figure 4.3**, it is seen that the average of the contig fragments for each neuron is generally centered around zero. This means, that even though these fragments come from one contig, the VAE is not able to detect a similar signal between the fragments from the same contig. This means that when averaging all the fragments for each neuron, the final latent space encoding for that particular contig would end up as zeros in most of the latent space neurons. There is, however, some of the neurons receives a signal which means that it is encoded away from zero.

4.2 Gumbel-softmax CVAE

In **Figure 4.4** it can be sees that when the temperature is lowered in the Gumbel-softmax, the reconstruction loss (CE loss) increases. This shows, that when the model is forced to make discrete predictions it does even worse than when it is not forced to do so. This is probably due to the way the loss is calculated. When the

model is making a high probability prediction which is false, the loss will increase dramatically compared to a model that makes 'vague' predictions. In other words it means that the model cannot learn how to reconstruct the input data since it generally has a low confidence in its predictions but is forced to 'guess'.

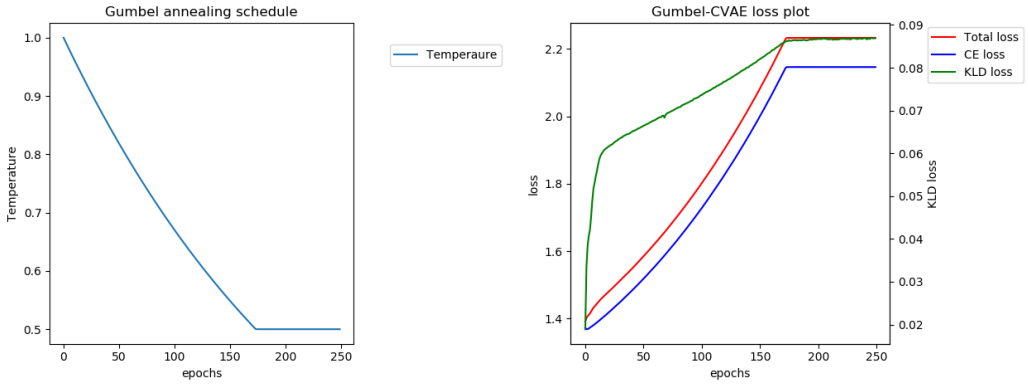

Figure 4.4: (a) shows how the temperature anneals. (b) Shows the loss of the model where the CE loss and KLD loss gradually increases due to the lower temperature

Furthermore, the clusters from the gumbel-softmax model was binned. Here it was found that there is actually less bins that can be detected, and all the bins are single contigs with a size $>$ 100000 (see **Table 4.2**). This means that the performance is actually worse for this model compared to the vanilla CVAE.

		Precision					
		0.3	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Recall	**0.3**	5	5	5	5	5	5
	0.5	2	2	2	2	2	2
	0.7	1	1	1	1	1	1
	0.9	1	1	1	1	1	1
	0.95	1	1	1	1	1	1
	0.99	1	1	1	1	1	1

Table 4.2: Number of bins generated by Gumbel-softmax CVAE with a minimum bin size of 100000.

4.3 CVAE with increased contig size

As described in the methods, a model with increased contig-fragment size was also created. This was done to make it easier for the model to cluster similar contigs

together because of the increased fragment length. The model that performed the best with the increased size had the following parameters:

Epochs	Batch size	Learning Rate	Latent Size	β	Dropout	Channels	Filters	Strides	Padding
250	1028	0.001	40	1000	0.3	[20,20,20,20,20]	[7,15,10,10,10]	[1,1,10,10,10]	[3,7,0,0,0]

A model with these parameter was test both using log-softmax (**Table 4.3**) and the Gumbel-softmax (**Table 4.4**) as described in the methods.

		Precision					
		0.3	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Recall	**0.3**	8	8	8	8	8	8
	0.5	5	5	5	5	5	5
	0.7	4	4	4	4	4	4
	0.9	4	4	4	4	4	4
	0.95	4	4	4	4	4	4
	0.99	2	2	2	2	2	2

Table 4.3: Number of bins generated by log-softmax CVAE with a minimum bin size of 100000 and minimum contig-fragment size of 10000.

		Precision					
		0.3	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Recall	**0.3**	5	5	5	5	5	5
	0.5	3	3	3	3	3	3
	0.7	2	2	2	2	2	2
	0.9	2	2	2	2	2	2
	0.95	2	2	2	2	2	2
	0.99	1	1	1	1	1	1

Table 4.4: Number of bins generated by Gumbel-softmax CVAE with a minimum bin size of 100000 and minimum contig-fragment size of 10000

Chapter 5

Discussion

For the different CVAE models there was created tables that show how many bins that was found. To actually compare how well the models performs on the Cami Airways dataset against other binners, it was decided to benchmark against VAMB. VAMB can process both contig TNF and abundance data. To make a fair comparison, the results was checked against a VAMB model using only contig TNF data. This means that both models use the same underlying data.

		Precision					
		0.3	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Recall	**0.3**	74	71	60	54	48	41
	0.5	58	56	47	43	39	33
	0.7	52	50	41	37	35	30
	0.9	36	34	25	22	20	18
	0.95	28	26	18	16	15	14
	0.99	20	19	12	11	10	10

Table 5.1: Number of bins generated by VAMB using contig TNF data

From **Table 5.1** it can be seen that VAMB outperforms all of the CVAE models in binning by a lot. The CVAE model that showed the best performance was the model that used log-softmax with a minimum contig-fragment size of 10000 (**Table 4.3**). This model was able to find 8 bins with a recall and precision ≥ 0.3, while VAMB found 74 with the same recall and precision.

Another thing to notice is how the number of contigs in the clusters that VAMB generates are distributed. When looking for clusters containing ≥ 50 contigs in them, it can be seen that there is a clear difference in the CVAE models compared

to VAMB (**Table 5.2**).

VAMB	Vanilla CVAE	Gumbel-softmax CVAE	CVAE fragment size 10000	CVAE fragment size 10000 Gumbel-softmax
104	10	7	6	5

Table 5.2: Number of clusters with ≥ 50 contigs in them for each model

VAMB	Vanilla CVAE	Gumbel-softmax CVAE	CVAE fragment size 10000	CVAE fragment size 10000 Gumbel-softmax
433	7263	3399	2451	2027

Table 5.3: Clusters with the highest number of contigs in them for each model

Furthermore, it can be seen in **Table 5.3** that the cluster with the highest number of contigs vary a lot from VAMB to the CVAE models. Combining the information from the two tables above, it can be inferred that the clusters generated by VAMB are more equally distributed, and there is not outliers, like in the CVAE models, where a few clusters contain a high amount of the contigs.

It is suspected that the reason for the CVAE models not performing very well is the fact that the contigs are split into fragments. When the contigs a fragmented there is a requirement for the model to be able to encode most of the fragments such that their average produces a signal that represents that contigs taxonomical profile. If this is not the case, similar contigs will not be clustered together. In **Figure 4.3** it can be seen that the contig fragments are for most neurons evenly distributed around zero. This figure shows it for one contig, however, the same can be seen if this is done with other contigs (see **Appendix A3**). Also, it was found that the neurons that are actually not averaged to zero are the same for the different contigs. When looking at **Figure 4.3** it can be seen that the neurons at the latent space dimension 14, 18 and 30 actually deviates from the other neurons. It was found that this is also the case for most of the other contigs. This could indicate that the CVAE model finds a weak signal in the contig-fragments where it is able to utilize a small number of the neurons in the latent space. But it also means that the full size of the latent space is not utilized, which shows that the overall signal from the data is weak.

The weak signal in the latent space contributed to the CVAE model not being able to

reconstruct the input. When looking at how the log-softmax models reconstructed the bases, it was found that it generally gave a probability of 0.25 for every base throughout the sequence length. This meant that the model could simply not learn how to reconstruct the input. To minimize the loss in this situation it puts the bases at an equal probability. To prevent this, the the Gumbel-softmax was introduced, this would force the model to make discrete predictions. However, it was found that when using the annealing schedule described in the methods and results, the Gumbel-softmax lead to a higher loss. The reason for this was that the model could still not reconstruct the input, and when it was forced to make discrete predictions that were wrong, it got penalized more than the log-softmax.

To further investigate the weak signal, 10 different contigs with a size $\geq$ 200000 all belonging to different organisms was picked. The contigs were splits into contig-fragments of size 2000 which meant that there was 100 fragments for each contig. After this, the TNF profile for every contig-fragment was computed. On this data, a PCA was performed if the PCA could find a signal in the fragments such that fragments belonging to the same contig would be clustered together (see **Figure 5.1**). When looking at the figure it can be seen that the contig-fragments are not clustered. This could indicate that using contig-fragments of size 2000, produces a signal that is too weak for the model to detect. It would also explain why the model with a contig-fragment size of 10000 produces a better result even though it discards a lot of data because of the size requirement.

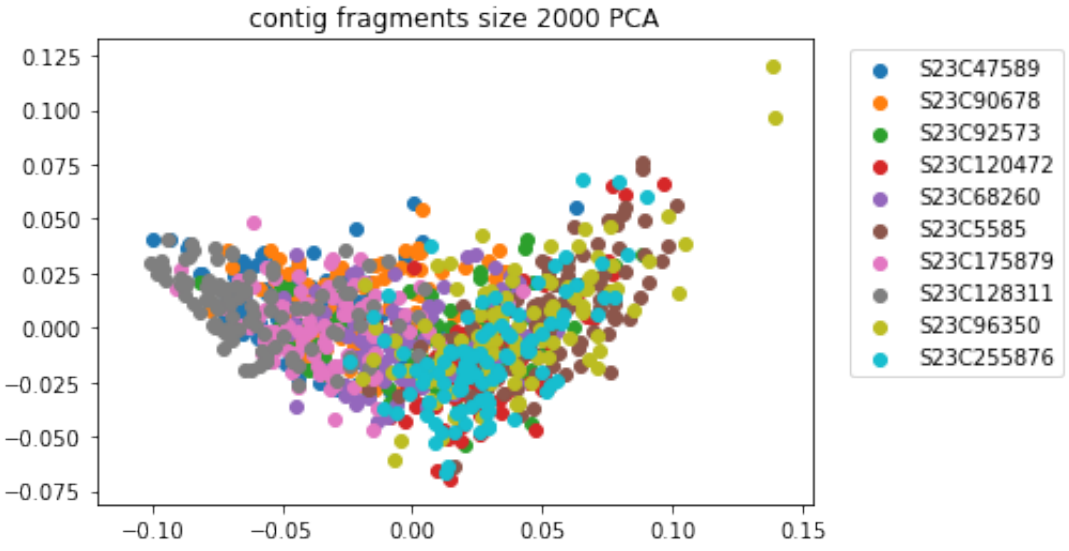

Figure 5.1: Principal Component Analysis on contig-fragments TNF profile.

The weak signal generated from the data also explains why there is a few clusters

with a huge amount of contigs in them (**Table 5.3**). Since all the contig-fragments are distributed around zero for every neuron in the latent space, many contigs will have the same encoding-profile which would put them in the same cluster when using the clustering algorithm. This means, that even though the contigs do not share any taxonomical relation they are still put in one huge cluster together. Another possible reason why VAMB outperforms the CVAE models could be that TNF reduces the overall noise in the data and extracts the most important information into a vector of size 136. Furthermore, the TNF profile for contigs that are not fragmented and therefore longer, will be more precise since it represents more of that organisms DNA.

For future work with creating a Variational AutoEncoder that directly uses sequence data, one could look into changing the convolutional layers to recurrent layers. Recurrent layers are able to compute over sequences of data with an undefined length. This would make it possible to not having to split the contigs, which causes the poor performance of the CVAE models. The biggest disadvantage of using recurrent layers is that they are computationally not feasible for large input data because of a lack of parallelization. To overcome this one could use a network type called a Quasi Reccurent Neural Network(QRNN)(Bradbury et al. 2016). QRRN's combine reccurent layers with convolutional layers to effectively compute over long sequences of data. This could fix the problem of fragmenting the contigs while still being computationally feasible. This could potentially lead to an effective model with interesting binning results.

Chapter 6

Conclusion

In this project, a convolutional Variational AutoEncoder was succesfully created that was able to compute on contig data. It was found that this model was not optimal for binning metagenomics data, due to the fact that contigs had to be split into fragments. When the contigs were split into fragments it was clear that the model had trouble with finding signal in the data, since fragments from the same contig was distributed around zero in the latent space for most of the latent space neurons. This meant that a few clusters with a lot of contigs were found, where the contigs in these clusters had no taxonomical relation. This affected our binning results such that the model had a poor performance when benchmarking against VAMB.

Appendix

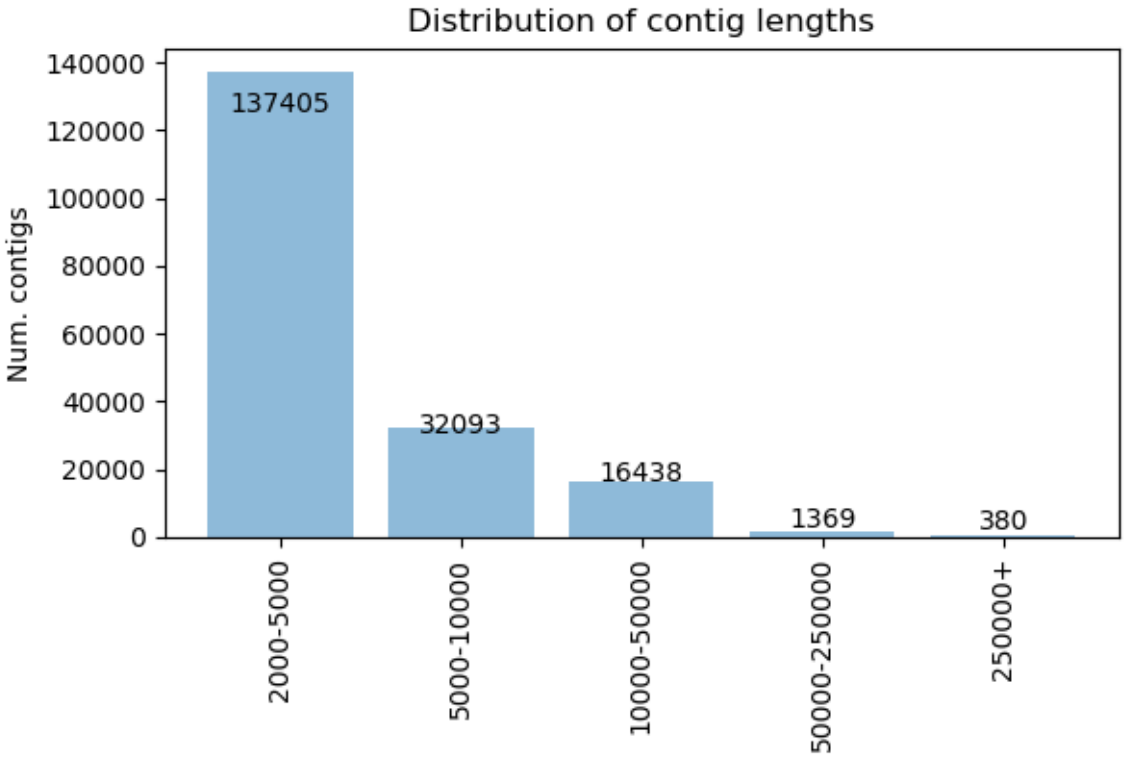

Figure A1: Bar chart showing distribution of contig lengths

Algorithm S1 Iterative medoid clustering

1: **function** CLUSTER($contigs, threshold, N_{samples}$)
2: $seed \leftarrow$ first contig of $contigs$
3: $cluster \leftarrow$ set of all contigs within $threshold$ of $seed$
4: $D(cluster) \leftarrow$ mean distance of $seed$ to each other member of $cluster$
5: **for** $trialseed$ of $N_{samples}$ randomly sampled contigs from $cluster$ **do**
6: $trial \leftarrow$ set of all contigs within $threshold$ of $trialseed$
7: $D(trial) \leftarrow$ mean distance of $trialseed$ to each other member of $trial$
8: **if** $D(trial) < D(cluster)$ **then**
9: $seed \leftarrow trialseed$
10: **go to** 3
11: **end if**
12: **end for**
13: **output** $cluster$
14: $contigs \leftarrow contigs \setminus cluster$
15: **if** $contigs \neq \emptyset$ **then**
16: **go to** 2
17: **end if**
18: **end function**

Supplementary Notes: Iterative medoid clustering algorithm used in the clustering step of VAMB.

Figure A2

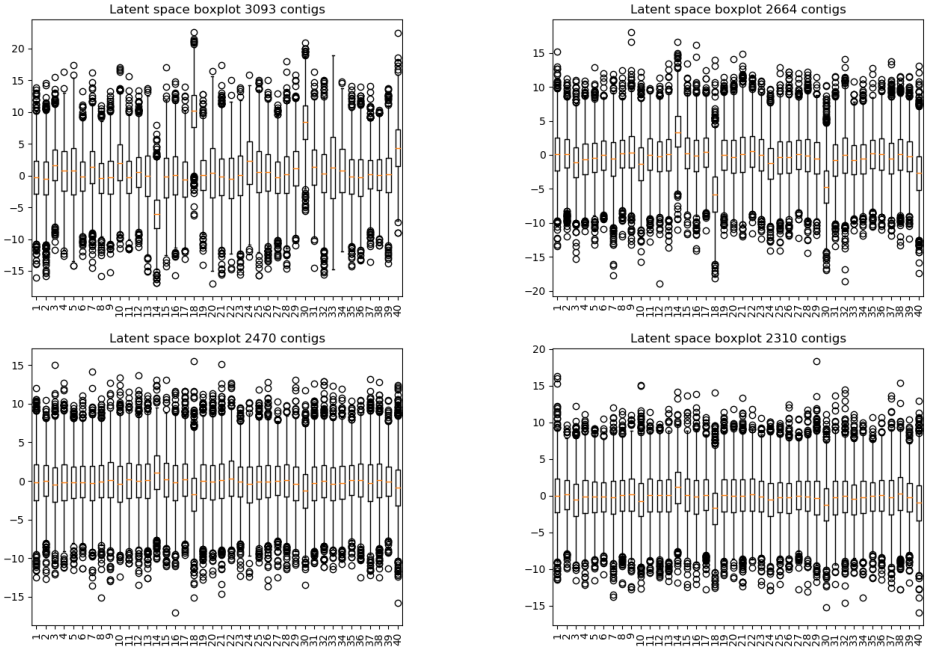

Figure A3: Boxplot of four different contigs showing how the contig-fragments are encoded. Generally it can be seen that the neurons in the latent space is averaged to zero which indicates a weak signal.

Bibliography

Bradbury, James et al. (2016). “Quasi-Recurrent Neural Networks”. In: (cited on page 30).

Doersch, Carl (2016). “Tutorial on Variational Autoencoders”. In: (cited on page 8).

Drmanac, Radoje (2011). “The advent of personal genome sequencing”. In: (cited on page 3).

Duchi, John and Elad Hazan (2011). “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: (cited on page 16).

Dumoulin, Vincent and Francesco Visin (2018). “A guide to convolution arithmetic for deep learning”. In: (cited on page 11).

Higgins, Irina et al. (2017). “β-VAE: LEARNING BASIC VISUAL CONCEPTS WITH A CONSTRAINED VARIATIONAL FRAMEWORK”. In: (cited on page 18).

Hinton, Geoffrey and Tijmen Tieleman (2012). “COURSERA; Neural Networks for machine learning Lecture 6a Overview of mini-batch gradient descent”. In: (cited on page 16).

Ioffe, Sergey and Christian Szegedy (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (cited on page 16).

Jang, Eric, Shixiang Gu, and Ben Poole (2016). “Categorical Reparameterization with Gumbel-Softmax”. In: (cited on page 18).

Kingma, Diederik P. and Jimmy Lei Ba (2015). “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”. In: (cited on page 16).

Kingma, Diederik P. and Max Welling (2014). “Auto-Encoding Variational Bayes”. In: (cited on page 8).

LeCun, Yann (1988). “A Theoretical Framework for Back-Probagation”. In: (cited on page 6).

LeCun, Yann et al. (1989). “Backpropagation applied to Handwritten Zip Code Recognition”. In: (cited on page 10).

Miranda, LJ (2017). “Understanding softmax and the negative log-likelihood”. In: (cited on page 17).

Nissen, Jakob Nybo et al. (2018). “Binning microbial genomes using deep learning”. In: (cited on pages 4, 19, 20).

Paszke, Adam et al. (2017). “Automatic differentiation in PyTorch”. In: (cited on page 15).

Pride, David T. et al. (2003). “Evolutionary Implications of Microbial Genome Tetranucleotide Frequency Biases”. In: (cited on page 12).

Quinlan and Hall (2010). “BEDTools: a flexible suite of utilities for comparing genomic features”. In: (cited on page 14).

Robbins, Herbert and Sutton Monro (1951). “A Stochastic Approximation Method”. In: (cited on page 6).

Rumelhart, D.E., Hinton, and R.J. Williams (1986). “Learning Internal Representations by Error Propagation”. In: (cited on page 7).

Saeed, Isaam, Sen-Lin Tang, and Saman K. Halgamuge (2011). “Unsupervised discovery of microbial population structure within metagenomes using nucleotide base composition”. In: (cited on page 12).

Sazli, Murat H. (2006). “A BRIEF REVIEW OF FEED-FORWARD NEURAL NETWORKS”. In: (cited on page 5).

Shlens, Jonathon (2005). “A Tutorial on Principal Component Analysis”. In: (cited on page 19).

Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: (cited on pages 16, 17).

Swirszcz, Grzegorz, Wojciech Marian Czarnecki, and Razvan Pascanu (2016). “Local minima in training of neural networks”. In: (cited on page 6).

Szegedy, Christian et al. (2015). “Going Deeper with Convolutions”. In: (cited on page 11).

Wang, Yi et al. (2012). “MetaCluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample”. In: (cited on page 12).