UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2251

MICROBE DETECTION USING DEEP LEARNING

Mirna Baksa

Zagreb, June 2020

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2251

MICROBE DETECTION USING DEEP LEARNING

Mirna Baksa

Zagreb, June 2020

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 13 March 2020

MASTER THESIS ASSIGNMENT No. 2251

Student: **Mirna Baksa (0036491078)**

Study: Computing

Profile: Computer Science

Mentor: prof. Mile Šikić

Title: **Microbe Detection Using Deep Learning**

Description:

Microbes are omnipresent organisms that are not possible to be seen by naked eye. The term includes many types of microorganisms, such as bacteria, viruses, fungi, etc. They affect and interact with humans in multiple ways - food production and digestion, immune system, and many other functions in the body. It is essential to be able to detect and classify them to discover diseases, prescribe medication, and keep a healthy lifestyle. The goal of this work is to detect microbe corresponding with the given sequenced DNA fragments by using a suitable representation of a sequenced signal. After finding a signal representation, the goal is to find an appropriate distance metric to separate different species in the latent space properly. In the end, reads will be classified using a classifier like k-NN. Develop a method for microbe detection based on a deep learning architecture. For evaluation of results, use a publicly available dataset, such as Zymo mock community dataset. The solution should be implemented in Python with the PyTorch or similar computational library. The source code should be documented using comments and should follow the Google Python Style Guide when possible. The complete application should be hosted on GitHub under an OSI approved license.

Submission date: 30 June 2020

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

DIPLOMSKI ZADATAK br. 2251

Pristupnica: **Mirna Baksa (0036491078)**
Studij: Računarstvo
Profil: Računarska znanost
Mentor: prof. dr. sc. Mile Šikić

Zadatak: **Prepoznavanje mikroba uporabom dubokog učenja**

Opis zadatka:

Mikrobi su sveprisutni, golim okom nevidljivi organizmi. Pod pojmom mikrob podrazumijevaju se mnoge vrste mikroorganizama poput bakterija, virusa, gljivica, itd. Oni utječu i međudjeluju s ljudima na više načina - u proizvodnji i razgradnji hrane, kod imunološkog sustava i kod mnogo drugih funkcija u ljudskom organizmu. Stroga, važno je moći prepoznati i klasificirati mikrobe kako bi otkrili bolesti, prepisali liječenje i zadržali zdrav način života. Cilj ovoga rada je prepoznavanje mikroba za dani očitani DNA fragment koristeći odgovarajuću reprezentaciju očitanog signala. Nakon reprezentacije signala, cilj je pronaći odgovarajuću metriku kako bi u latentnom prostoru uspješno razdvojili različite mikrobe. U konačnici, očitanja klasificirati koristeći klasifikacijski model, npr. k-NN. Zadatak je razviti metodu za otkrivanje mikroba koristeći metode dubokog učenja. Za evaluaciju može se koristiti neki javno dostupni skup podataka, primjerice Zymo mock community skup podataka. Rješenje je potrebno implementirati u programskom jeziku Python koristeći PyTorch ili sličnu biblioteku za matrični izračun. Izvorni kod je potrebno dokumentirati koristeći komentare i razvijati prema Google Python Style Guide kada je to moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednu od OSI odabranih licence.

Rok za predaju rada: 30. lipnja 2020.

I would like to express my sincere gratitude to my mentor Mile Šikić for his constant motivation, extensive guidance and opportunities given throughout my entire university experience.

An additional thank you to Dominik Stanojević for his help with this thesis.

Last, but not least, I thank my family and friends for their unconditional love, support and encouragement, without which my success would not have been possible.

CONTENTS

1. Introduction **1**

2. Dataset **3**

2.1. Preprocessing 4

2.1.1. Raw Signals 4

2.1.2. Reducing dimensionality 6

2.1.3. Normalization 7

2.1.4. Summary 8

2.2. Zymo mock community dataset 8

3. Methods **10**

3.1. Overview 10

3.1.1. Artificial Neural Networks 11

3.1.2. Convolutional Neural Networks 13

3.1.3. Recurrent Neural Networks 14

3.1.4. Long Short-Term Memory Networks 15

3.2. Models 17

3.2.1. Autoencoders 17

3.2.2. Triplet networks 22

3.3. Visualization 23

3.4. Classification 24

3.5. Technical Stack 25

4. Results **27**

4.1. Autoencoders 29

4.1.1. Convolutional Autoencoder 30

4.1.2. LSTM Autoencoders 35

4.1.3. Autoencoder Performance Summary 40

4.2. Triplet Networks 41
4.2.1. Convolutional Triplet Network 43
4.2.2. LSTM Triplet Network 48
4.2.3. Further Experiments 51
4.2.4. Triplet Networks Performance Summary 55

5. Evaluation **56**
5.1. Results on the Zymo mock community dataset 56
5.2. Discussion 58

6. Conclusion **59**

Bibliography **61**

List of Figures **63**

List of Tables **66**

1. Introduction

Microbes are microscopic organisms composed of either a single cell, multiple cells or cell clusters. The most common types include viruses, bacteria, protozoa, and fungi.

These microorganisms live in almost every part of the biosphere: in the soil, deserts, the ocean floor, high in the atmosphere, and deep within the Earth, thus adapting to most conditions - including extremes such as very high or cold temperatures, high pressure or even high radiation environments.

Microbes interact with human culture in many ways. The human body is home to millions of microorganisms in the normal body flora, but some microbes are also pathogens responsible for infectious diseases. Outside of the body, microbes are a contributor to treating sewage, producing enzymes and fuel, fermenting various foods (e.g. beer or wine), and much more.

Studying microbial genomes helps to better understand their biological components and how their genetic configuration contributes to their distinct characteristics. The field of microbial genetics has advanced tremendously with DNA sequencing, the process of determining the order of nucleotides in a DNA fragment. First proposed by Frederick Sanger in 1977, sequencing technology has since gone through many iterations of improvement, where the ideal sequencer would be highly accurate, with long read length (no gaps in the genome), low cost, and high throughput.

Nanopore sequencing is a third generation sequencing technology promising very long reads, high throughput, and low material requirement. A nanopore is essentially a very small (<1nm in width) hole through which DNA strands are driven by electrophoresis. As each base molecule is driven through the nanopore, it induces a different change in current which is then used to identify that particular molecule. These raw current signals outputted by the sequencer are the target of this work. The challenge is that the DNA strands move very rapidly through the nanopore which makes the signals prone to background noise.

Working with sequences, such as the raw nanopore signals, is one of the harder challenges in machine learning and data science industries. Real-life sequences are un-

predictable, containing complex temporal dependencies, where genomes as sequences are particularly complicated in their nature. When considering the added noise, one can easily see why this problem is especially hard. Another thing to keep in mind that a DNA strand is formed by only four types of nucleotides whose order is what determines the genetic code, so there is not much diversity in the type of molecule passing through the nanopore and thus inducing a change in current.

The goal of this thesis is to research different methods for finding a compressed signal representation for a DNA fragment of a microbe. The compressed representation should contain the most important features of the fragment so that different microbial species can be distinguished according to their representations. The end objective is to be able to cluster different DNA fragment signals according to their species and therefore be able to easily classify a new, unseen, and unknown DNA fragment. If the method is good enough, the compressed representations could be used for different purposes as they would contain the most important information from the genome, while being easier to work with because of their reduced dimensionality.

In this thesis, chapter 2 gives an introduction to the dataset and the preprocessing flow. Chapter 3 will give a theoretical overview of methods and architectures used in this work, then introduce the exact models used, visualization and classification techniques, and lastly a brief description of the technical stack. Chapter 4 presents and discusses the results. Chapter 5 discusses the evaluation on true data, with a conclusion in chapter 6.

2. Dataset

The starting point for this work are genomes from 6 different species:

- *Escherichia coli*
- *Bacillus anthracis*
- *Klebsiella pneumoniae*
- *Pantonea agglomerans*
- *Pseudomonas koreensis*
- *Yersinia pestis*

Original genome lengths are shown in table 2.1. During nanopore sequencing, each nucleotide in the genome produces some change in current so the raw signal is even longer.

Table 2.1: Original Genome Lengths

Species	**Total Length (bp)**
Escherichia coli	4,641,652
Bacillus anthracis	5,227,293
Klebsiella pneumoniae	5,682,322
Pantonea agglomerans	5,115,241
Pseudomonas koreensis	6,301,761
Yersinia pestis	4,829,855

These reference genomes are long, complex, and very similar to each other. The similarity between genomes is higher when organisms are closer in a phylogenetic tree. The actual similarity of prokaryotic genomes can be calculated using the Average Nucleotide Identity (ANI) score. Given two genomes, the ANI score estimates nucleotide identity between their coding regions. For example, ANI score $\geq 95\%$ is usually a

boundary for same species genomes. ANI scores calculated with ORTHOAni (Lee et al. (2016)) [1], are shown in table 2.2.

Table 2.2: Average Nucleotide Identity scores

Bacillus anthracis	62.2472%				
Klebsiella pneumoniae	78.3507%	61.9479%			
Pantonea agglomerans	73.4330%	61.7770%	74.0607%		
Pseudomonas koreensis	66.4474%	62.5534%	67.3194%	66.5931%	
Yersinia pestis	71.1529%	62.885%	72.2456%	71.7984%	65.2331%
	Escherichia coli	**Bacillus anthracis**	**Klebsiella pneumoniae**	**Pantonea agglomerans**	**Pseudomonas koreensis**

2.1. Preprocessing

Each reference genome is split into smaller sections, each containing around 10k nucleotides. The sections are generated using a sliding window with an offset of 1000. The sliding window concept (on a smaller scale). is shown in figure 2.1 .

Using this technique, for a genome of approximately 5 million base pairs long around 5k sections are generated. Once the sections are generated, they are transformed into raw electrical current signals using DeepSimulator.

This process is reversed from what happens in practice, where raw nanopore signals are the starting point, but this is a way of generating an artificial dataset to work on.

2.1.1. Raw Signals

A tool called DeepSimulator by Li et al. (2018) is used to reproduce the raw current signals. While most existing simulators simulate the reads based on statistical patterns of data, DeepSimulator imitates the flow of the nanopore sequencing procedure.

[1] https://www.ezbiocloud.net/tools/orthoani

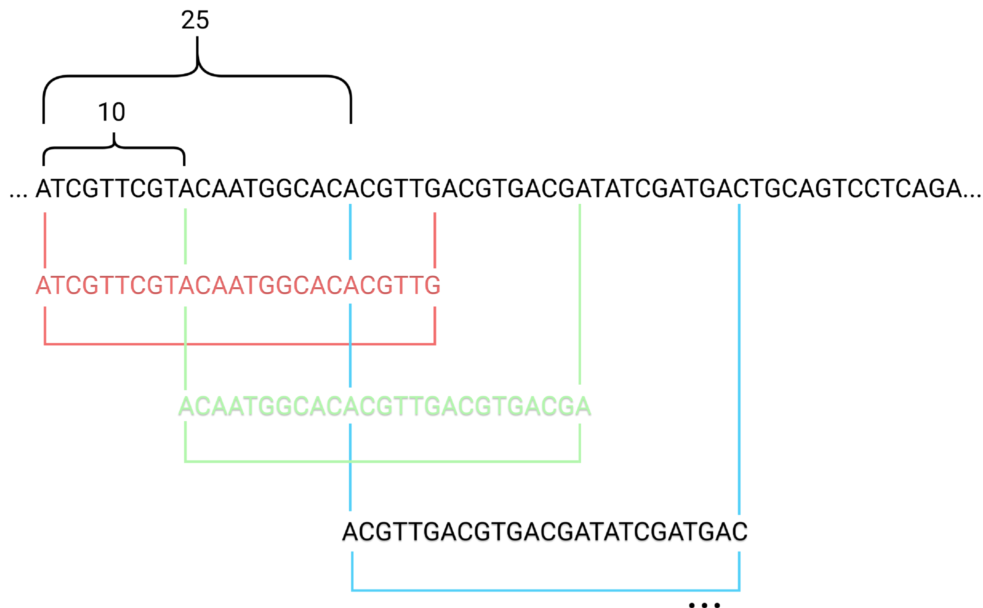


Figure 2.1: Sliding window concept with window length 25 and offset 10

Taking in a reference genome, DeepSimulator generates raw electrical current signals using a context-dependent deep learning model. Simulated reads are yielded by base-calling.

To generate a signal, the user needs to provide a reference genome and specify either the number of reads or the coverage. The sequence is preprocessed before going into the signal generation model. The signal generation model generates the expected electrical signal of a particular *k*-mer (usually 5-mer or 6-mer).

In terms of this work, DeepSimulator with default parameters is used to simulate the raw current signals. Reads generated by DeepSimulators default parameters are bound to have almost exact characteristics as the actual data.

A sample output (the first 1000 signal timesteps) is shown in figure 2.2.

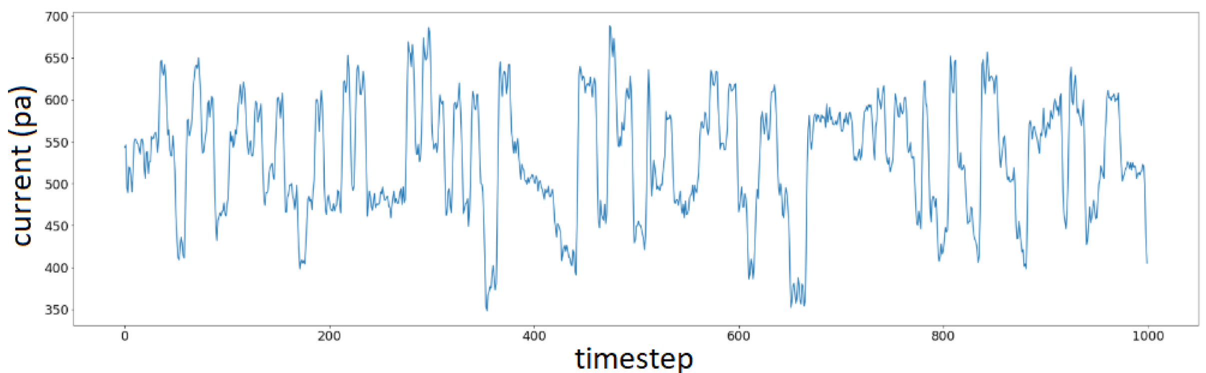


Figure 2.2: Example of the first 1000 timesteps of a signal generated by DeepSimulator

2.1.2. Reducing dimensionality

After retrieving the signals from DeepSimulator, the dimensionality of data is large. To reduce, the input samples are split again into fragments of 400 timesteps (of the raw signal) which is equivalent to 40-50 base pairs of the reference genome. This division is arbitrary - fragment length is chosen so that the end result is as representative as possible, but of lower dimensionality.

The fragment of length 400 is then represented by 4 different statistical coefficients: 2 measures of central tendency - the **mean** and the **median**:

$$Mean = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{2.1}$$

$$Median = \begin{cases} \frac{(N+1)^{th}}{2} term; \text{ when } n \text{ is } even, \\ \frac{\frac{N^{th}}{2}\text{ term}+(\frac{N}{2}+1)^{th}\text{ term}}{2}; \text{ when } n \text{ is } \textit{odd} \end{cases} \tag{2.2}$$

and 2 measures of variability - the **standard deviation** and the **interquartile range**:

$$Standard\ Deviation = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{2.3}$$

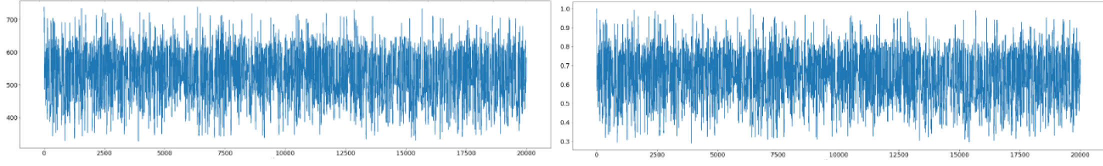
$$Interquartile\ Range = Q_3 - Q_1 = 3(\frac{N+1}{4})^{th}\ term - (\frac{N+1}{4})^{th}\ term \tag{2.4}$$

This way, a fragment of the sample of length 400 is mapped to 4 statistics, so the starting input sample goes from shape $(timesteps, 1)$ to $(timesteps/400, 4)$.

The dimensionality of the dataset is thus significantly reduced while the 4 descriptive statistics coefficients hopefully still retain enough information about the features of the signal.

The pipeline for 20k timesteps can be seen in figures 3.1 and 2.4.

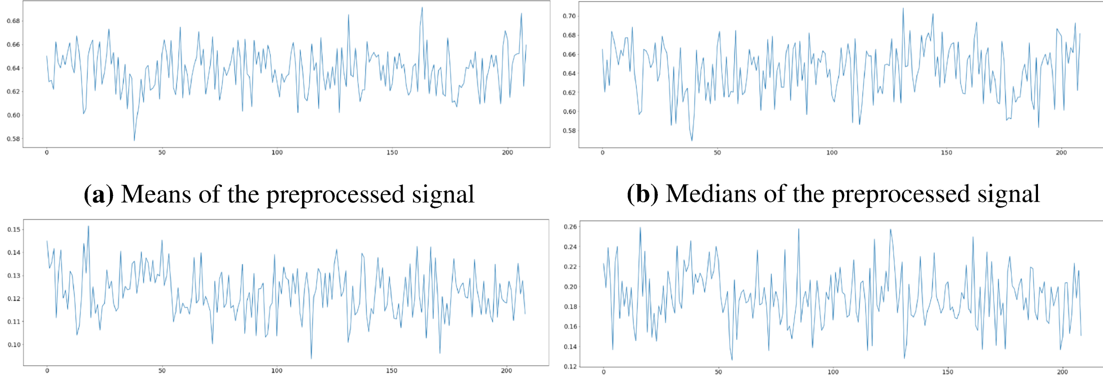
This dataset is split evenly between the training/validation/test in the ratio 80% / 10% / 10 %.



(a) Example of a raw signal

(b) Example of a raw signal scaled to $(0, 1)$

Figure 2.3: Raw signal example



(a) Means of the preprocessed signal

(b) Medians of the preprocessed signal

(c) Standard Deviations of the preprocessed signal

(d) Interquartile Ranges of the preprocessed signal

Figure 2.4: Preprocessed signal example. A raw signal is first split into fragments of 400 timesteps which are then each mapped to 4 statistics - mean, median, standard deviation, interquartile range.

2.1.3. Normalization

The final step is **normalizing** the inputs. Each of the datasets (train, validation, and test) is normalized separately so as not to introduce any bias from the training set into the validation and test sets.

The inputs are in shape

$$\boldsymbol{x} = \begin{pmatrix} mean_0 & median_0 & stdev_0 & iqr_0 \\ mean_1 & median_1 & stdev_1 & iqr_1 \\ \dots & \dots & \dots & \dots \\ mean_n & median_n & stdev_n & iqr_n \end{pmatrix} \tag{2.5}$$

Normalizing by **standard score** is done by calculating the **mean** μ and **standard deviation** σ of every column of every input sample - the mean of all means, the standard deviation of all means; the mean of all medians, the standard deviation of all medians etc.

Normalization is then done as

$$x' = \frac{x - \mu_{column}}{\sigma_{column}} \tag{2.6}$$

where x' is the normalized value of original value x.

Another way of normalizing is by **min-max scaling**, calculating the **max** and **min** of each column and computing

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.7}$$

Both of these ways are used in this work, each for an appropriate network implementation.

2.1.4. Summary

To sum up, table 2.3 shows the total number of input samples for each class in the dataset and the percentage of samples belonging to that class.

Table 2.3: Sample distribution through classes

Species	Total samples	Percentage in dataset
ecoli	4632	14.64%
bacillus anthracis	5218	16.49%
klebsiella pneumoniae	5636	17.81%
pantonea agglomerans	5078	16.04 %
pseudomonas koreensis	6292	19.88 %
yersinia pestis	4792	15.14 %

The ideal case would be 16.67% ($\frac{1}{6}$) of samples for each class which would mean a perfectly balanced dataset. This dataset is not exactly perfectly balanced, but is quite close to it - no class deviates from the 16.67% for more than 3%.

2.2. Zymo mock community dataset

Once a successful architecture is found, it will be evaluated on actual signal data from the **Zymo mock community** publicly available dataset[2] by Loman et al. (2018).

The ZymoBIOMICS Microbial Community Standard is the first commercially available standard for studies in metagenomics and microbiomics. Mock community standards are useful for the development and validation of not just laboratory but software and bioinformatics methods as well.

[2]https://github.com/LomanLab/mockcommunity

This particular community consists of ten microbial species: eight equally distributed bacteria (12% each): *Escherichia coli, Pseudomonas aeruginosa, Salmonella enterica, Enterococcus faecalis, Lactobacillus fermentum, Staphylococcus aureus, Listeria monocytogenes* and *Bacillus subtilis*; and two yeasts (each present at 2%): *Saccharomyces cerevisiae* and *Cryptococcus neoformans*.

The whole dataset consists of:

- **4.23M** reads
- **16.59Gb** bases
- **4.620bp** read length N50.

Since the dimensionality of this raw signal dataset is very large, the evaluation in the scope of this work will be done on a smaller, manageable part of that dataset. The dataset is originally given in 43 batches but only reads from the first batch are processed (around 200k reads).

Similar to many metagenomic samples, the exact species is unknown at the time of sequencing, meaning that the raw signals from the dataset are not assigned to a specific class. This posed a new problem since this work requires labelled data.

To mend this problem, Kraken 2 (Wood et al. (2019)) was used as a taxonomic sequence classification system. Kraken 2 is an improved version of Kraken (originally presented by Wood and Salzberg (2014)). The classifier maps original k-mers within the reference to the lowest common ancestor, or LCA, in all existing genomes containing that particular k-mer. Along with the sequenced Zymo dataset, a Kraken microbial community database was published publicly as well[3]. This database was used to classify the base-called reads which were then cross-referenced with signal files to get a dataset of classified raw signals. The process is not exact and probably not 100% accurate since the Kraken and cross-referencing pipeline introduces some space for unintentional error.

Raw signals from 6 of the 10 species (to match the artificial dataset) were preprocessed in the same manner as the artificial dataset, with 4000 reads chosen from each species - giving in total 24 000 reads to process.

The evaluation on a smaller scale dataset is still not a real-life scenario but will give some indication on how well the developed model works.

[3]https://lomanlab.github.io/mockcommunity/mc_databases.html

3. Methods

3.1. Overview

The main objective of this work is to research an appropriate deep architecture that could generate a compressed representation (encoding) of a given signal. The representation should be such that different species can be segregated by their representations. The basic pipeline of this work is shown in figure 3.1.

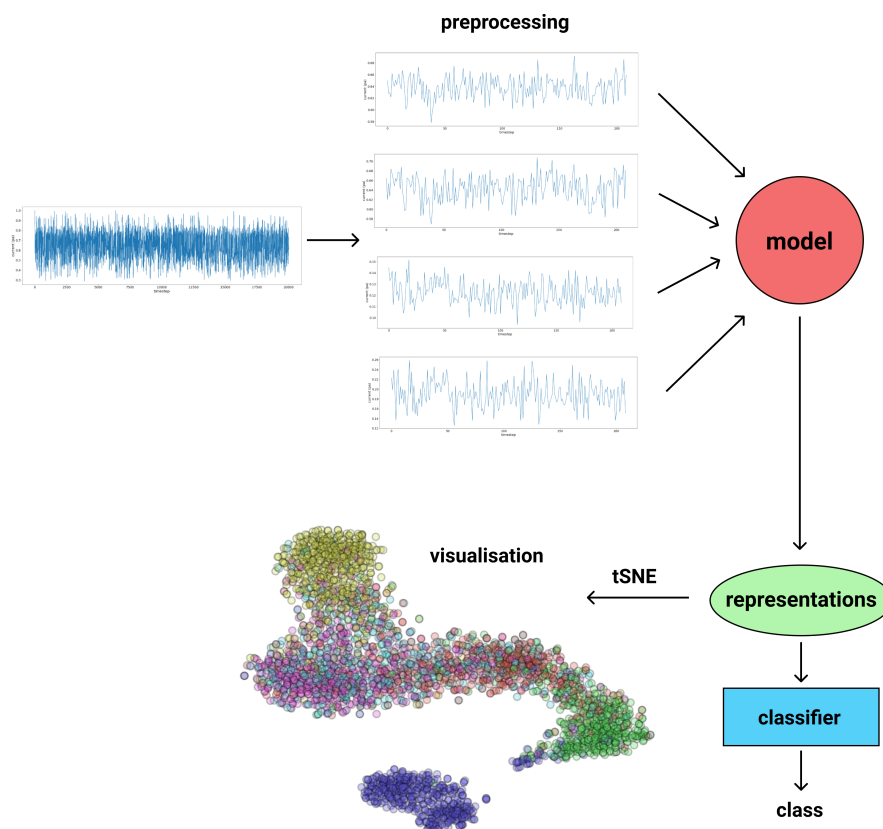


Figure 3.1: Scheme of the work pipeline. Raw signals go through a preprocessing stage before they are fed into a model. The model learns signal representations, which are visualized with tSNE and classified to a microbial species.

After the preprocessing stage described in section 2.1, the signals are fed into the

chosen model. An overview of the models used is shown in figure 3.2.

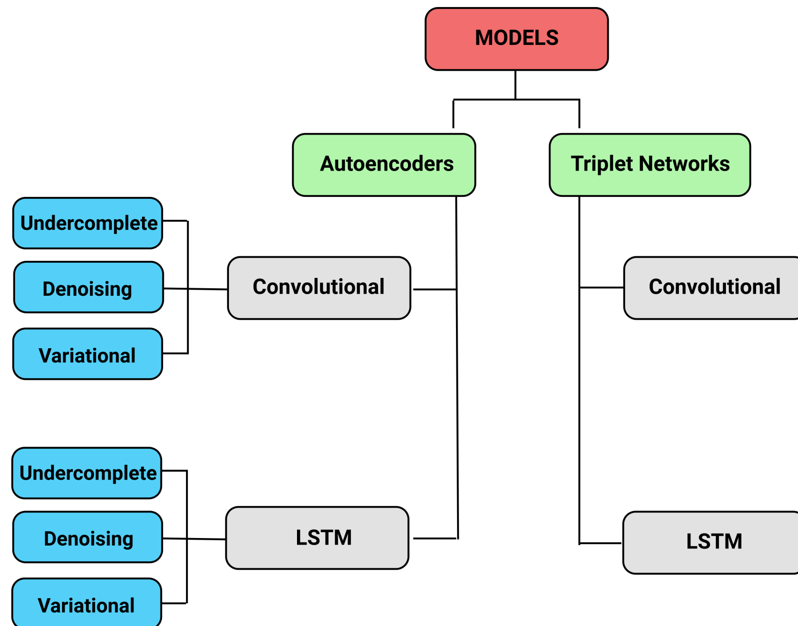


Figure 3.2: Overview of models used in this work

After training the model, the signals are visualized. The first indicator of the efficiency of the model is how well the representations of different classes are clustered together. A more concrete metric is given by the classifier which is the last step of the pipeline.

The remainder of this section is a brief overview of each type of architecture used and how it can be adjusted to the problem at hand.

3.1.1. Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational networks inspired by biology, built of interconnected units called neurons. ANNs are capable of learning non-linear functions, thus called **universal function approximators**.

Neural networks comprise three layers: the input layer, hidden layers (one or more), and an output layer. A neural network consisting of one hidden layer is illustrated in figure 3.3.

The network weights are updated by the **backpropagation** algorithm which computes the gradient of a pre-determined loss function in relation to the network weights for an input-output sample (or a batch of samples). The gradient is calculated using the chain rule, calculating layer by layer, starting from the last one. The network learns

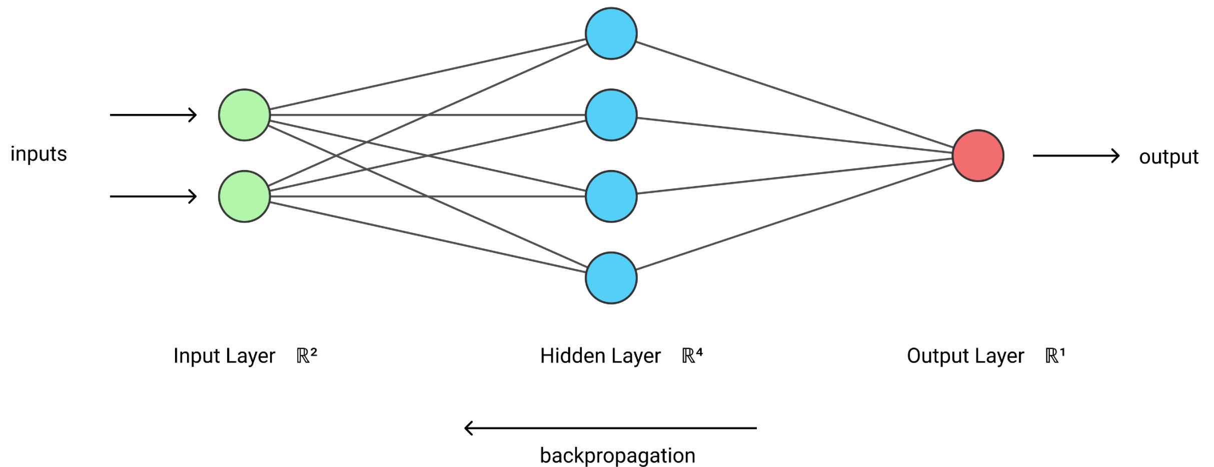


Figure 3.3: Simple Artificial Neural Network architecture with an input layer, one hidden layer and an output layer

by updating its weights according to the calculated gradient, which is a measure of the change in weights compared to the change in error.

Problems that emerge in deep (with more layers) neural networks are the **exploding** and **vanishing gradient** problems. As already mentioned, the gradient has a direct effect on how well the network is learning.

In some cases this gradient is small - many traditional network activation functions (functions that define the output of a given node) have gradients $\in (0, 1)$. When applying the chain rule on an n-layer network, a small gradient will result in effectively multiplying n small numbers to calculate the gradients of the front layers. This causes the gradient to decrease exponentially with n, which has a negative effect on the training of the front layers of the network. This is the so-called **vanishing gradient problem**.

A demonstration of the problem can be seen in figure 3.4. The figure shows the effect of applying a sigmoid activation function multiple times. The function is flattened until it has no slope, which is equivalent to a gradient vanishing when passed through multiple layers.

On the other hand, the **exploding gradient problem** occurs when activation function gradients take on larger values. In that case gradients accumulate quickly which results in an unstable network.

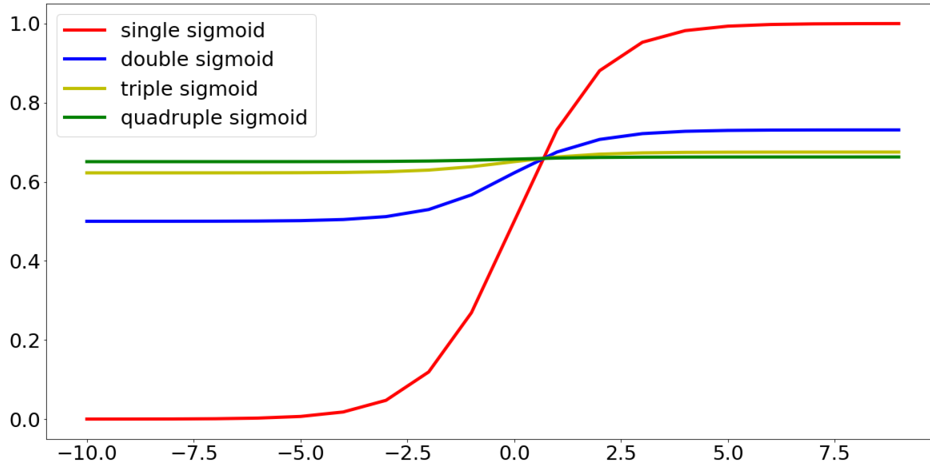


Figure 3.4: Effect of applying the sigmoid activation function multiple times

3.1.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are specialized for processing data with grid-like topology. The most common use cases are for image processing seeing that images are essentially 2D grids of pixels, but they are also used in time-series data which can be presented as a 1D grid.

Where other neural networks use regular matrix multiplication, CNNs employ an operation called **convolution**:

$$s(t) = (x * w)(t) \tag{3.1}$$

In CNN terminology, the x is often referred to as the **input**, w as the **kernel** and the output as the **feature map**.

Although colloquially referred to as convolutions, CNNs technically actually employ a **cross-correlation** (a sliding dot product). Its effect on a 1D grid is shown in figure 3.5.

The intention of the convolution operation is to extract high-level features from the input.

Fully connected neural networks are able extract some features as well, but only with a very large amount of neurons. This is because of the matrix multiplication in fully connected neural networks where every output unit interacts with the input. The convolution operation mends this problem - computing the output requires fewer operations, the model has to store fewer parameters which both reduces the memory usage of the model and improves its efficiency.

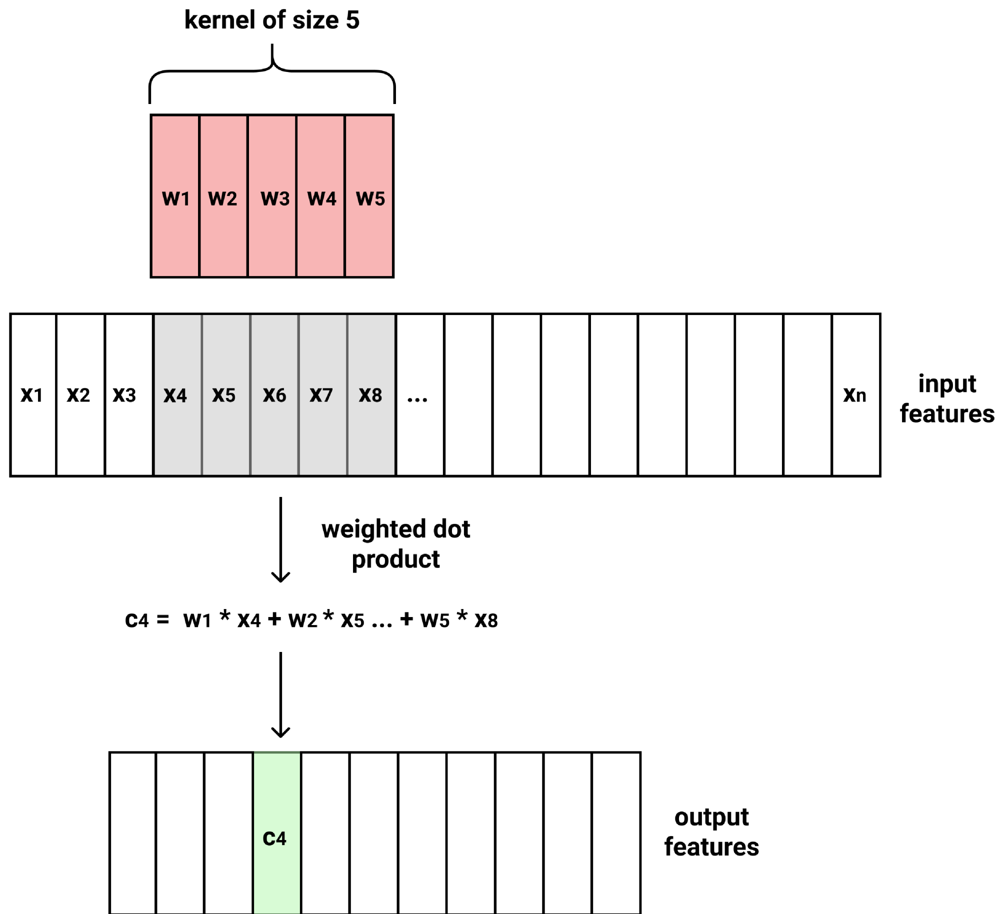


Figure 3.5: Cross-correlation (convolution) applied to a 1D grid

3.1.3. Recurrent Neural Networks

The problem with simple ANNs and sequential data is that their architecture is such that they can not capture temporal information in data and therefore underperform when dealing with sequences.

An improvement on simple artificial neural networks are recurrent neural networks which are designed to be able to capture dependencies in sequential data. Recurrent networks take as input the current input example, and additionally their outputs from previous inputs by a feedback loop, as in figure 3.6. Therefore, recurrent networks have two input sources: the present and the recent past.

Because of their architecture, recurrent networks are often characterized to have memory, mathematically described as

$$h_t = \theta(Wx_t + Uh_{t-1}) \tag{3.2}$$

Memory helps in dealing with sequences - information about the sequence gets preserved in the hidden state of the recurrent network.

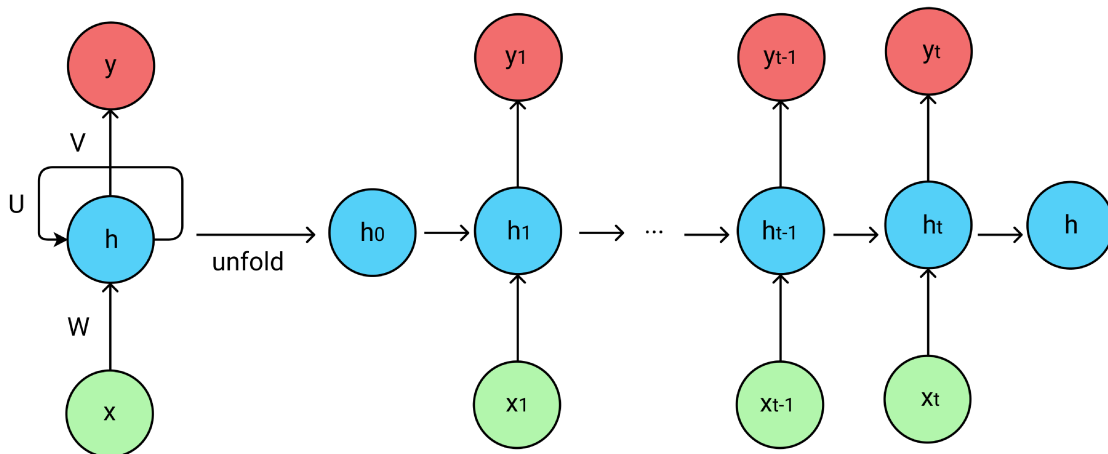


Figure 3.6: Recurrent Neural Network architecture

RNNs are not immune to the vanishing and exploding gradient problems. Since recurrent networks seek correlation between points many timesteps apart, a vanishing gradient will make that impossible for complex problems. It can be said that RNNs suffer from short-term memory - if a sequence is long enough, a RNN will struggle with carrying information from early steps to later ones.

3.1.4. Long Short-Term Memory Networks

Long Short-Term Memory, (LSTM) Networks were proposed by Hochreiter and Schmidhuber (1997) and are a subtype of recurrent neural networks. LSTMs were developed as a solution to the already described vanishing gradient problem.

LSTM networks proved to perform better than conventional feed-forward neural networks and RNNs for processing time-series data where there can be a delay between important events in the sequence. Some applications include machine translation, speech recognition, future prediction (e.g. predicting stock prices), etc.

In terms of the vanishing gradient problem, LSTMs help maintain constant error and preserve backpropagated error through time and layers. This is done by storing information in a gated cell as shown in figure 4.24.

The LSTM gated cell learns when to allow changes of the cell state through gates that control the information flow. The gates filter information with their own set of weights which are adjusted during the learning procedure through gradient descent, just like any other weight in a neural network. When the cell makes a decision about what needs to be learned it effectively decides which parts of a sequence are important to remember when processing a long chain of sequences.

Formally, each LSTM cell will compute the following for each element in the input

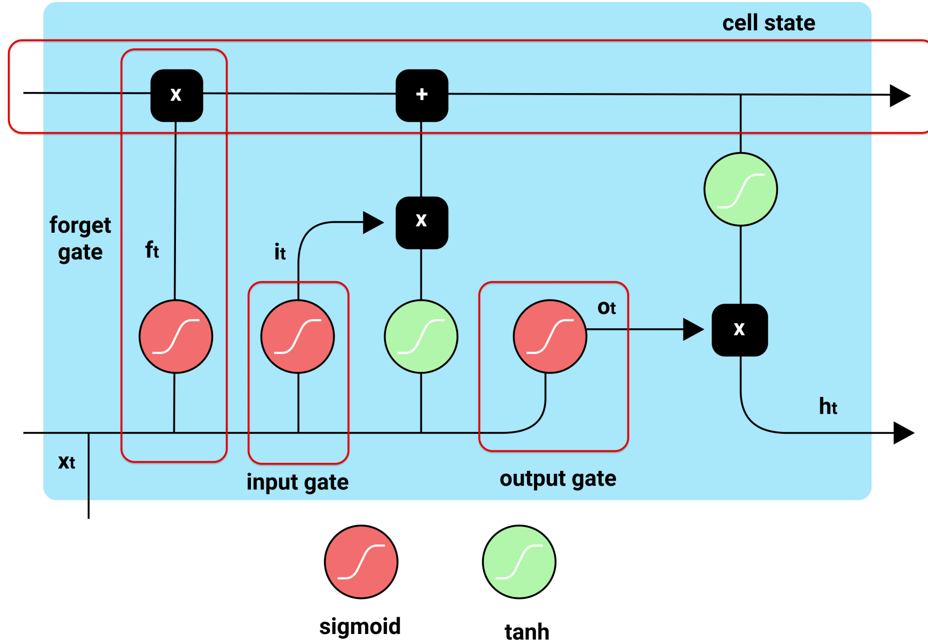


Figure 3.7: LSTM cell scheme

sequence:

$$
\begin{aligned}
i_t &= \delta(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \delta(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \delta(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot TanH(c_t)
\end{aligned}
\tag{3.3}
$$

where h_t is the hidden state at the final moment t, c_t is the cell state at t, x_t is the input at t, h_{t-1} is the layer hidden state at $t-1$ or the initial hidden state at 0, b is the bias and i_t, f_t, g_t, o_t are the input, forget, cell and output gates. δ is the sigmoid function and $\odot$ represents the Hadamard product.

The key advantage of LSTM units over traditional RNN neurons is that LSTMs sum activities over time, which means that the derivative will not vanish as quickly and long-range features are easier to discover.

Bidirectional LSTMs

An extension to the traditional RNN is the bidirectional RNN proposed by Schuster and Paliwal (1997). This idea has been used extensively on LSTMs.

Bidirectional LSTM Networks train two instead of one LSTM on the input sequence. The first LSTM input is the sequence as-is, while the second LSTM gets a time-reversed replica of the input sequence. By doing this, the network has more context on the given sequence.

The simplest example is speech - sounds, words and sentences can at first mean nothing, but as future context is provided they start to make sense. Bidirectional RNNs and LSTMs do exactly that - provide both current and future context. Although for some problems using a bidirectional network may not be suitable (e.g. online learning where one requires an output after every input), in others they do offer some benefits and improve performance.

3.2. Models

3.2.1. Autoencoders

An autoencoder is a neural network whose objective is to learn data encodings. The autoencoder architecture is designed so that it imposes a bottleneck in the network which encourages it to extract meaningful features and produce a compressed representation of the input.

The general structure of an autoencoder network is presented in figure 3.8.

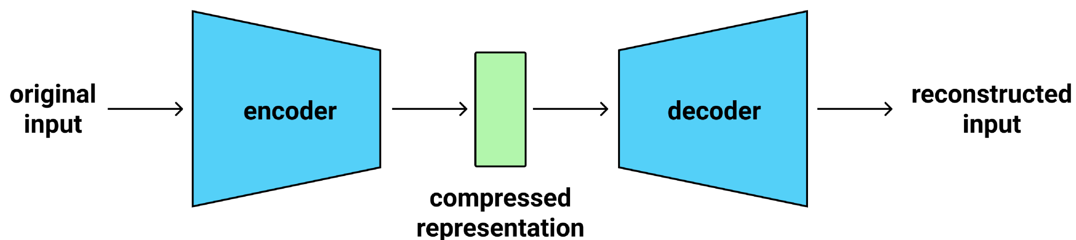


Figure 3.8: Autoencoder architecture, consisting of an encoder which learns compressed representations, and a decoder which learns to map the representation back to the original output

Two main parts of an autoencoder are the encoder and the decoder. The encoder learns how to reduce the input dimensions and encodes the input data into a compressed representation. This can be expressed as a function $h = f(x)$, where h is the previously mentioned bottleneck in the network - the compressed representation. The decoder then tries to reconstruct the data from the compressed representation, or mathematically $r = g(h)$. What needs to be avoided is an autoencoder simply learning to set $g(f(x)) = x$ which is not very useful.

Autoencoders are usually optimized by minimizing the reconstruction loss

$$\mathcal{L}(in, out) = \mathcal{L}(x, g(f(x))) \tag{3.4}$$

which calculates the difference between input and its reconstruction. Such losses are e.g. the mean absolute error (also called L1) or the mean squared error:

$$\begin{aligned} MAE &= \frac{1}{n}\sum_{t=1}^{n}|e_t| = \frac{1}{n}\sum_{t=1}^{n}|in_t - out_t| \\ MSE &= \frac{1}{n}\sum_{t=1}^{n}e_t^2 = \frac{1}{n}\sum_{t=1}^{n}(in_t - out_t)^2 \end{aligned} \tag{3.5}$$

The basic architecture is as shown in figure 3.8, but the actual implementation can vary depending on the use case. An autoencoder could be a simple feed-forward network, a convolutional network, an LSTM, etc.

Autoencoders are usually restricted to reconstruct the input approximately in order not to directly duplicate the input signal. This ensures that the compressed encoding retains only the most significant features of the given data.

Undercomplete autoencoders

The simplest way of preventing the autoencoder to repeat the input as the output is by constraining the dimensions of the network bottleneck - i.e. constraining the dimension of the encoding to be less than the dimension of input data.

Unfortunately, in some cases this is still not enough and an autoencoder with enough capability will be able to learn an identity function. As described by Goodfellow et al. (2016), one could in theory imagine a powerful autoencoder with a one-dimensional code where the encoder could learn to represent a training sample x_i with the code i. A decoder could then learn to map those integers back to specific training examples. Although this most likely will not happen in practice, it is a good illustration of an autoencoder failing to learn anything useful from the data.

Denoising autoencoders

The basic idea of denoising autoencoders is to train a network which extracts useful information by changing the reconstruction error.

As already mentioned, autoencoders minimize the function

$$\mathcal{L}(\boldsymbol{x}, g(f(\boldsymbol{x}))) \tag{3.6}$$

where $\mathcal{L}$ is a loss function punishing $g(f(\boldsymbol{x}))$ when different from x. A denoising autoencoder on the other hand will minimize

$$\mathcal{L}(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}}))) \tag{3.7}$$

where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ corrupted by noise.

The noise forces the autoencoder to remove the corruption instead of copying the input. As shown by Alain and Bengio (2012), a denoising autoencoder with small noise corruption can aid in f and g successfully learning the actual structure of the data.

Variational autoencoders

A variational autoencoder (VAE), Kingma and Welling (2013) and Jimenez Rezende et al. (2014), is an autoencoder with additional constraints on the encoded representations.

More precisely, instead of learning arbitrary encoding functions, variational autoencoders optimize the parameters of a distribution which models the input. This fact makes a variational autoencoder a generative model - when the autoencoder learns a probability distribution of some data, points can be sampled from that distribution.

Statistical motivation

Let there exist a hidden variable z which generates an observation x as in figure 3.9.

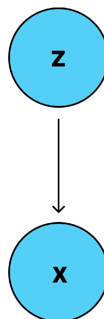


Figure 3.9: Hidden variable z generating an observation x

The aim is to infer z (in this case, the latent representation of x) by only knowing x.

Introducing the Bayes Rule:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{3.8}$$

where $p(z|x)$ is the posterior distribution over z, $p(x|z)$ is the likelihood function of z and $p(z)$ is the prior probability of z. Computing the posterior distribution is known as the **inference** problem.

The problem is that

$$p(x) = \int p(x|z)p(z)dz \tag{3.9}$$

can be very high-dimensional and difficult to compute. To estimate this value, $p(z|x)$ can be approximated with another distribution $q(z|x)$ such that its distribution is tractable. As shown in figure 3.10, the observations will be used to estimate the hidden variable. If the parameters of $q(z|x)$ are very similar to $p(z|x)$, q can be used to approximately infer the intractable distribution p.

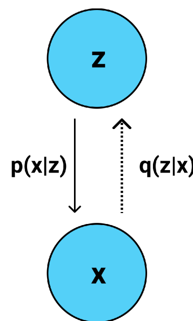


Figure 3.10: Using known observations x to estimate the hidden variable z

The autoencoder will now look as in figure 3.11

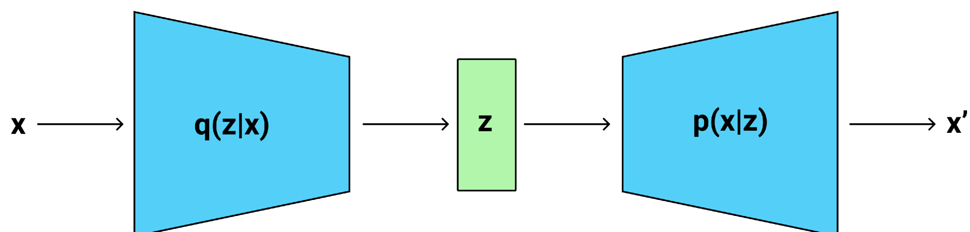


Figure 3.11: Variational Autoencoder architecture: $p(x|z)$ is approximated with a distribution $q(z|x)$

To measure the difference between p and q, **Kullback-Leibler divergence** (also called relative entropy) is used. Kullback-Leibler divergence (KL) is a measure of how one probability distribution is different from another, computed as

$$D_{\mathrm{KL}}(P||Q) = \int_{\infty}^{\infty} p(x)\log(\frac{p(x)}{q(x)})dx \tag{3.10}$$

The model is trained using a loss function that is a sum of two functions: the usual reconstruction loss and the Kullback–Leibler divergence. Conceptually, this means that one term of the loss will penalize the usual reconstruction error, while the second term will encourage the distribution $q(z|x)$ to be as close as possible to the prior distribution $p(z)$, for which the Gaussian distribution is assumed.

$$\mathcal{L}(x, x') + \sum_{j} \mathrm{KL}\left(q_j(z|x) || \, p(z) \right) \tag{3.11}$$

To implement the statistical motivation, the variational autoencoder conceptually will be as shown in figure 3.12.

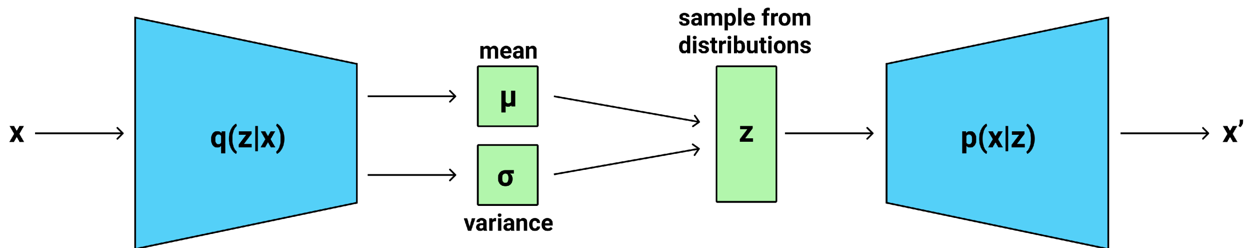


Figure 3.12: Variational Autoencoder implementation: encoder maps the input into μ and σ in latent space. A similar point z is sampled from $\mathcal{N}(\mu, \sigma)$, and fed to the decoder

In a concrete implementation, an encoder maps each input sample x into two parameters in latent space - mean value μ and variance σ. These vectors define the multivariate normal distribution around the input point. Then, a similar point z is randomly sampled from that distribution and returned as the latent variable. Finally, a decoder network is used to map these latent variables to original data.

The above-described pipeline makes sure that the network learns a smoother representation - a small change in the latent variable will not cause the decoder to produce a largely different output because the points are sampled from the same continuous distribution.

Reparametrization trick

In their original form, variational autoencoders sample from

$$z \sim \mathcal{N}(\mu, \sigma) \tag{3.12}$$

The sampling process is random. When training a model by backpropagation, the relationship of each network parameter to the network output needs to be calculated which can not be done for a randomly sampled parameter.

To solve this problem, the **reparametrization trick** suggests to introduce a new parameter ε, randomly sampled from a unit Gaussian $\sim \mathcal{N}(0, 1)$. ε is then shifted by the latent distribution mean μ and scaled by the latent distribution variance σ. The trick is shown in figure 3.13.

The reparametrization trick enables random sampling from the latent distribution, while still being able to infer the actual distribution parameters.

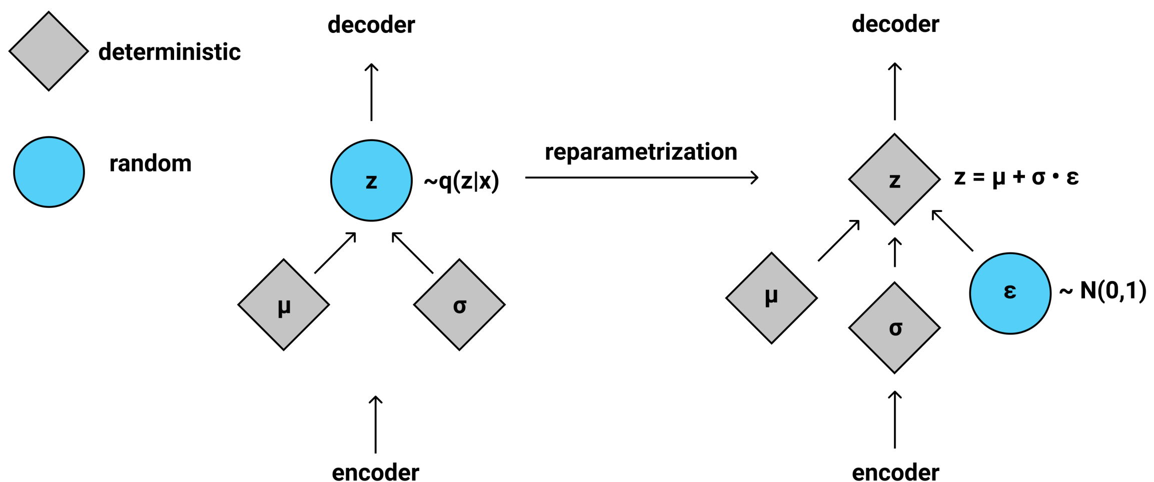


Figure 3.13: The effect of the reparametrization trick on random sampling: introducing a new parameter ϵ randomly sampled from a unit Gaussian enables random sampling of z

3.2.2. Triplet networks

The triplet concept, first proposed by Hoffer and Ailon (2014) is a way to perform unsupervised feature learning. Although not another neural network architecture (rather a way of training), this concept can be leveraged to retrieve encoded representations.

A triplet network is a neural network of arbitrary architecture (i.e. convolutional, LSTM...) trained using triplets $(\boldsymbol{x}, \boldsymbol{x}^+, \boldsymbol{x}^-)$ such that

- $\boldsymbol{x}$ is an arbitrary **anchor** sample
- $\boldsymbol{x}^+$ is a **positive** sample semantically similar to x (i.e. same class as x)
- $\boldsymbol{x}^-$ is a **negative** sample semantically dissimilar to x (i.e. different class than x)

The network is trained to minimize the loss defined as

$$\mathcal{L} = \boldsymbol{max}(\boldsymbol{d}(anchor, positive) - \boldsymbol{d}(anchor, negative) + margin, 0) \tag{3.13}$$

which pushes $d(anchor, positive)$ to 0, and $d(anchor, negative)$ to be greater than $d(anchor, positive) + margin$, where $d(p, q)$ is an arbitrary measure of distance. The goal is to push positive examples to be closer to the anchor and negative examples further from it, as presented in figure 3.14.

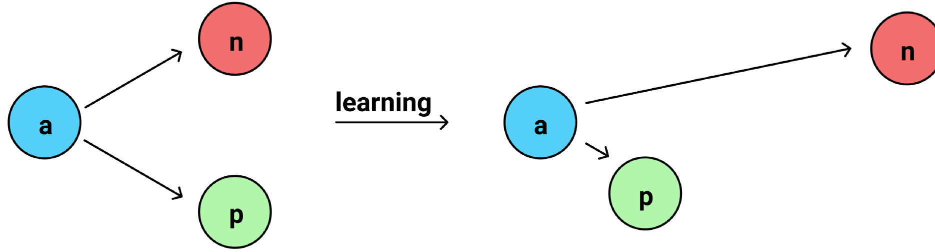


Figure 3.14: Triplet learning: positive samples are pushed closer to the anchor and negative samples further from it

Triplet networks were popularized further by Google's FaceNet (Schroff et al. (2015)) where triplet loss was used to learn an embedding space for face images.

The application of triplet networks to this work is simple - the input samples pass through the network and the loss will minimize the distance between embeddings of the same class (anchor embedding and positive embedding), and maximize the distance between embeddings of the opposite class (anchor embedding and negative embedding).

$$\mathcal{L} = \boldsymbol{max}(\boldsymbol{d}(emb_{anchor}, emb_{positive}) - \boldsymbol{d}(emb_{anchor}, emb_{negative}) + margin, 0) \tag{3.14}$$

The distance measure used is a simple Euclidean distance.

$$d\left(p, q\right) = \sqrt{\sum_{i=1}^{n} \left(q_i - p_i\right)^2} \tag{3.15}$$

3.3. Visualization

For visualization, the **T-distributed Stochastic Neighbor Embedding (t-SNE)** algorithm is used. Developed by Van der Maaten and Hinton (2008), this technique is especially suitable for high-dimensional data visualization.

The way this algorithm works is that it minimizes the divergence between two distributions: the distribution of pairwise similarities of the inputs and the distribution of

pairwise similarities of corresponding low-dimensional embedding points. Essentially, the algorithm tries to represent high-dimensional data by using fewer dimensions while still matching both distributions.

Clusters given by t-SNE plots can be misleading, depending heavily on the chosen algorithm parametrization. Nonetheless, in this work t-SNE is used simply as a visual aid - concrete metrics are calculated on the original embeddings.

All of the result images in chapter 4 are generated using t-SNE.

3.4. Classification

Once suitable signal representations exist, a method for their successful classification is needed. The simplest solution is the k-nearest neighbors algorithm.

K-nearest neighbors

K-nearest neighbours (*k*-NN) is probably the most straightforward classifier among many existing machine learning methods.

K-NN works in two phases - it will first determine the nearest neighbours of an unseen sample and then use them to infer the class of the sample using those neighbours. The simplest idea of a two-class problem in two-dimensional feature space is depicted in figure 3.15, but it can be similarly applied to a *n*-class problem in *n*-dimensional space.

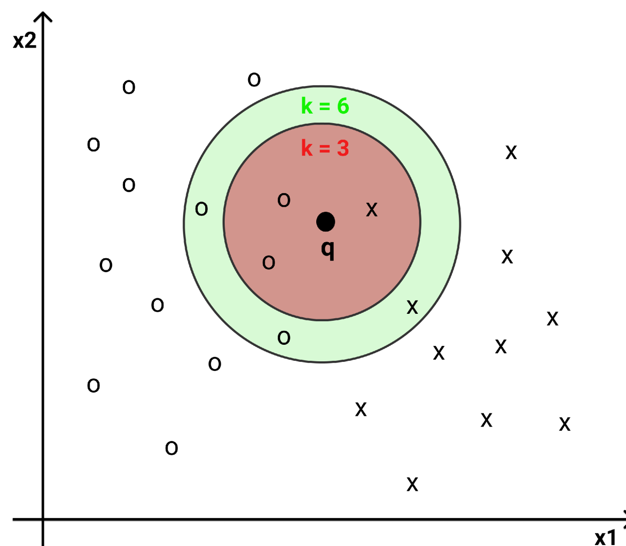


Figure 3.15: K-nearest neighbors algorithm idea in 2D feature space: the class of q is determined by the classes of k nearest neighbors

The classification decision for sample q can be made with majority voting or distance weighted voting based on the classes of the nearest neighbours.

Formally, let

$$D = \{(\boldsymbol{x_i}, y_i), i = 1, ..., n_D\} \tag{3.16}$$

be the training dataset of observed data, where the vector $\boldsymbol{x_i} = (x_{i1}, ..., x_{ip})$ represents predictor values and y_i denotes class membership.

For a new observation (q, y_q) the nearest neighbour x is determined by a distance metric:

$$d(q, x) = min_i d(q, \boldsymbol{x_i}) \tag{3.17}$$

with the distance metric typically (but not exclusively) being the Euclidean distance:

$$d\left(p, q\right) = \sqrt{\sum_{i=1}^{n} \left(q_i - p_i\right)^2} \tag{3.18}$$

The *k*-nearest neighbours are selected based on the given distance metric. This ends the algorithms first stage.

To determine the class of the new observation in the second phase, let *K* be the set of *k*-nearest neighbours of the given sample. Its class is then:

$$h(q) = \underset{j \epsilon (0,...,y_n)}{\arg\max} \sum_{(x^i, y^i) \epsilon K} \kappa(x^i, q) 1\{y^i = j\} \tag{3.19}$$

where κ is the kernel function assigning weights to the neighboring points based on some measure of distance to the query point. For simple majority voting, this function always gives 1.

3.5. Technical Stack

The solution is implemented using the **PyTorch** computational library.

PyTorch is a Python-based scientific computing package with two-fold usage - tensor computation (like NumPy) with GPU acceleration and a deep learning research platform with maximum flexibility and speed.

As a step further, the PyTorch Lightning library as a high-level PyTorch wrapper was used for organizing and automating PyTorch code. Lightning organizes the code

in 3 categories: research code, engineering code, and non-essential code such as logging. By refactoring to PyTorch Lightning: no flexibility is lost; no more unnecessary boilerplate code; the code is generalizable and adaptable to any hardware; readability and reproducibility are improved; advanced logging possibilities.

PyTorch Lightning enables automatic logging to TensorBoard - TensorFlow's visualisation toolkit, which was heavily used in this work to save experiment parameters, calculate metrics, plot training graphs, and plot resulting images.

All experiments were done on 2 GPUs.

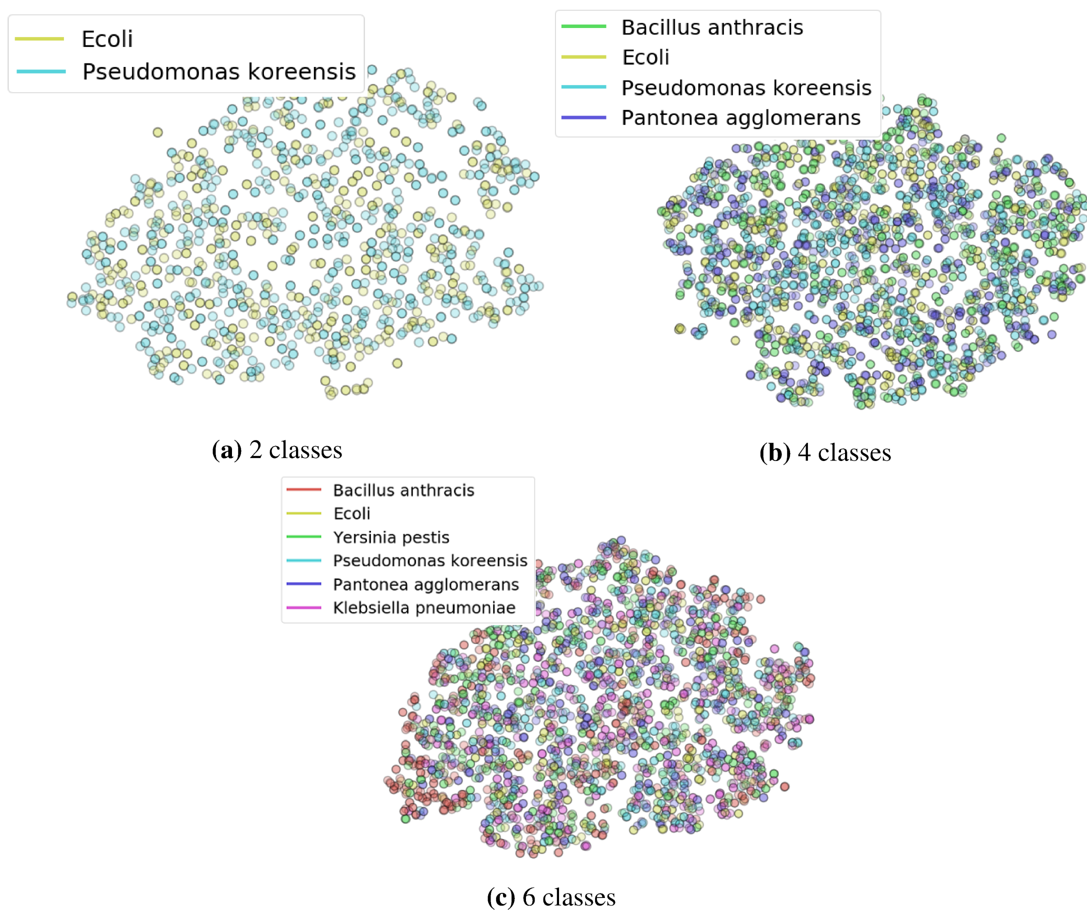
All of the code is publicly available on GitHub [1].

[1] https://github.com/mirnabaksa/Masters-Thesis

4. Results

This chapter shows the results of all experiments done on this dataset. Each model architecture was tested first on 2 classes, then 4 and in the end with 6 classes, under the assumption that the model complexity has to grow with the number of classes.

For reference, figure 4.1 shows the test set representations before training. The goal is to get somewhat distinct clusters for each class.



(a) 2 classes

(b) 4 classes

(c) 6 classes

Figure 4.1: Test set representations before training, visualized with t-SNE

Multiple architectures were tested, each with different configurations until the best result was found. All experiments were trained until overfit and then tested with the

model state just before the point of overfitting to get the best results. Some parameters were kept constant across all experiments as shown in table 4.1.

Table 4.1: Constant Parameters

Constants	
optimizer	adam
reconstruction loss	L1
k	$\sqrt{\text{\# test samples}}$

Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm which computes individual learning rates for different parameters. Empirically, Adam performs well in practice and compares favourably to similar stochastic optimization methods.

The value of k listed in table 4.1 refers to the hyperparameter of the k-NN classification algorithm. Finding the right value of k is not trivial - a large value of k is computationally expensive, whereas the noise will have a higher influence on the final result with a small k. The convention is to start out at $k = \sqrt{N}$, where N is the total number of samples, and optimize from there. Since the emphasis of this work is more on the models, k was kept constant at that value.

Overview

A short summary for context before presenting the results is that autoencoder architectures (both convolutional and LSTM, undercomplete, denoising, and variational) did not give satisfactory results.

Even when the signal was more or less reconstructed, the learned signal encodings did not manage to detect any clusters of data. The architectures were tested out on dummy datasets as well (e.g. with classes of very similar sine and cosine waves) and achieved successful separation of clusters there.

Triplet networks were much more successful in learning meaningful signal encodings.

4.1. Autoencoders

To best see the results of training autoencoders, the input, output, and target sequences (from the test set) were plotted. The input and target sequence will be the same in variational autoencoders, while in denoising autoencoders the input contains random noise, as shown in figure 4.2. In order not to overflow this work with images, only the sequence of the means (as described in chapter 2.1) will be plotted as a representative.

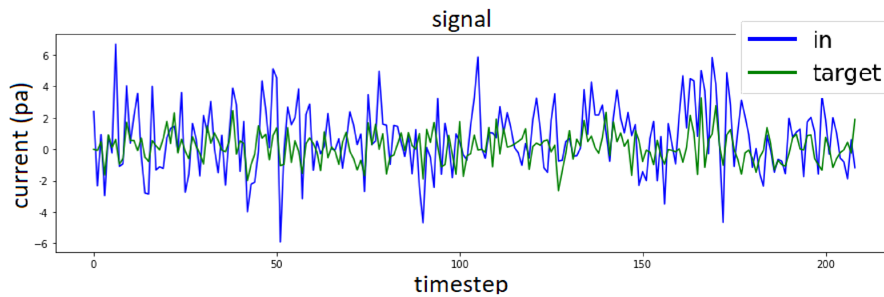


Figure 4.2: Input sequence with added Gaussian noise and target sequence

Some experiments on autoencoders which did not yield satisfactory results were left out of this section, mostly not to saturate the work with images of unsuccessful architectures.

Dataset

The sizes of the split dataset are shown in table 4.2.

Table 4.2: Autoencoder dataset sizes

	2 classes	**4 classes**	**6 classes**
Training set	8 739	16 635	25 318
Validation set	1 092	2 079	3 165
Test set	1 093	2 080	3 166

4.1.1. Convolutional Autoencoder

Undercomplete

2 classes

Results for this experiment are shown in figures 4.3 and 4.4, parameters in table 4.3a and metrics in table 4.3b. Network structure is shown in table 4.4.

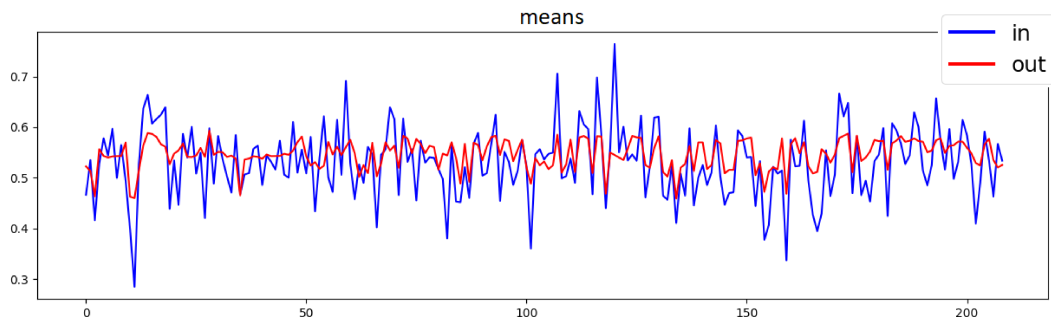
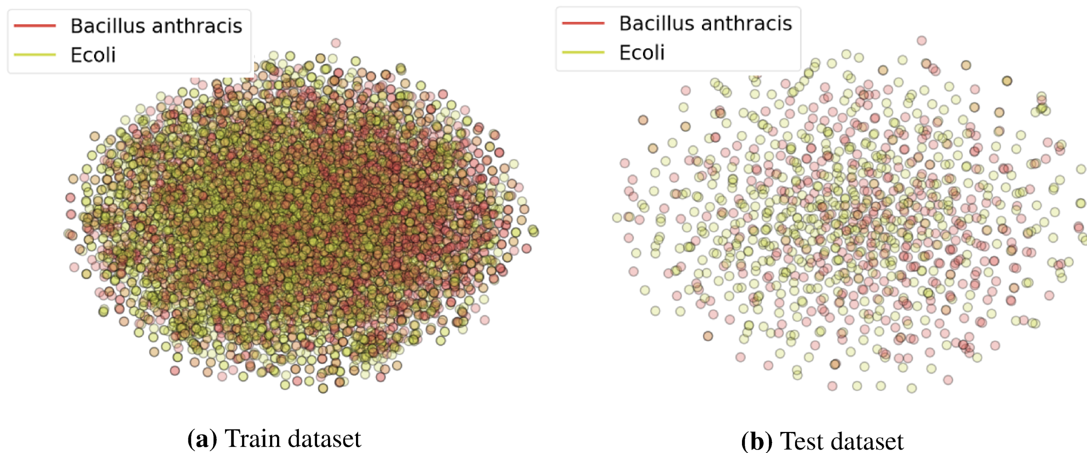


Figure 4.3: Undercomplete Convolutional Autoencoder output on 2 classes



(a) Train dataset

(b) Test dataset

Figure 4.4: Undercomplete Convolutional Autoencoder results on 2 classes, visualized with t-SNE

Denoising

4 classes

Results for this experiment are shown in figures 4.5 and 4.6, parameters in table 4.5a and metrics in table 4.5b. Network structure is shown in table 4.6.

Table 4.3: Undercomplete Convolutional Autoencoder experiment on 2 classes

Parameters	
epochs	1000
encoding size	42

(a) Parameters

Metrics	
accuracy	0.5756

(b) Metrics

Table 4.4: Undercomplete Convolutional Autoencoder structure

Layer name	No. of filters	Kernel size	Activation
encoder			
Conv1D	16	3	ReLU
Conv1D	32	3	ReLU
Conv1D	64	3	ReLU
Conv1D	1	5	ReLU
decoder			
Conv Transpose 1D	64	5	ReLU
Conv Transpose 1D	32	3	ReLU
Conv Transpose 1D	16	3	ReLU
Conv Transpose 1D	4	3	Tanh
Linear	timesteps		

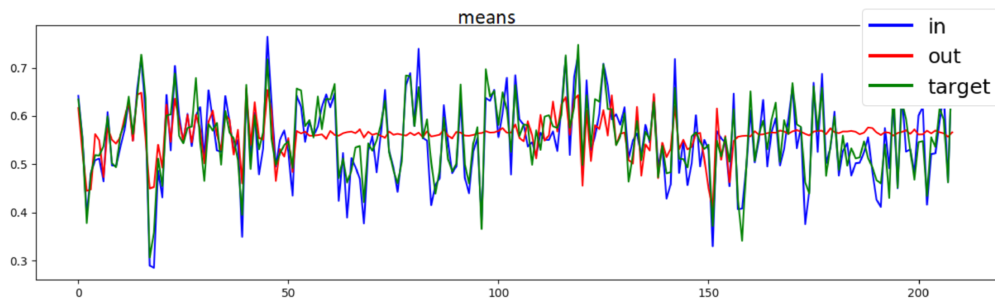


Figure 4.5: Denoising Convolutional Autoencoder output on 4 classes

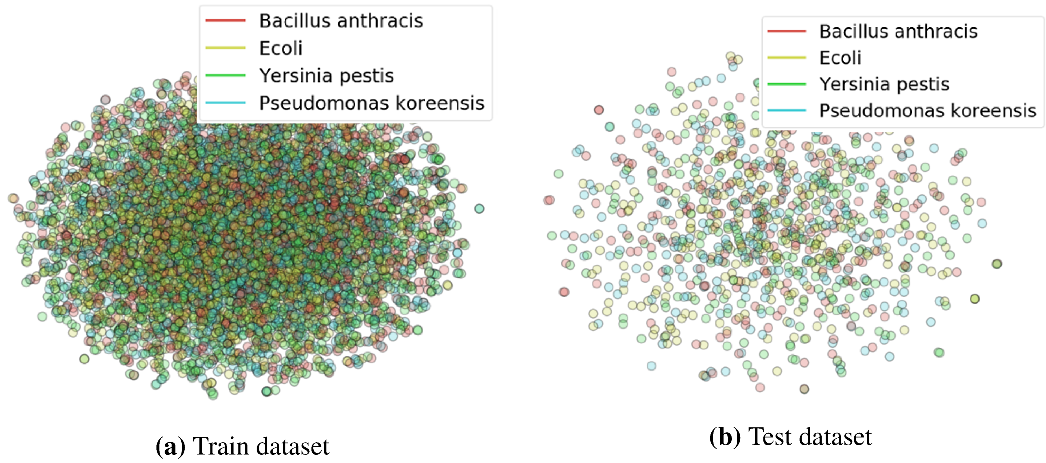
Table 4.5: Denoising Convolutional Autoencoder experiment on 4 classes, visualized with t-SNE

Parameters	
epochs	1200
encoding size	64

(a) Parameters

Metrics	
accuracy	0.33

(b) Metrics



(a) Train dataset

(b) Test dataset

Figure 4.6: Denoising Convolutional Autoencoder results on 4 classes, visualized with t-SNE

Table 4.6: Denoising Convolutional Autoencoder structure

Layer name	No. of filters	Kernel size	Activation
encoder			
Conv1D	16	3	ReLU
Conv1D	32	3	ReLU
MaxPool 1D			
Conv1D	64	3	ReLU
Conv1D	32	3	ReLU
MaxPool 1D			
Conv1D	16	3	ReLU
Conv1D	1	7	ReLU
decoder			
Conv Transpose 1D	16	7	ReLU
Conv Transpose 1D	32	3	ReLU
Conv Transpose 1D	64	3	ReLU
Conv Transpose 1D	32	3	ReLU
Conv Transpose 1D	16	3	ReLU
Conv Transpose 1D	4	3	Tanh
Linear	timesteps		

Variational

2 classes

Results for this experiment are shown in figures 4.7 and 4.8, parameters in table 4.7a and metrics in table 4.7b. Network structure is shown in table 4.8.

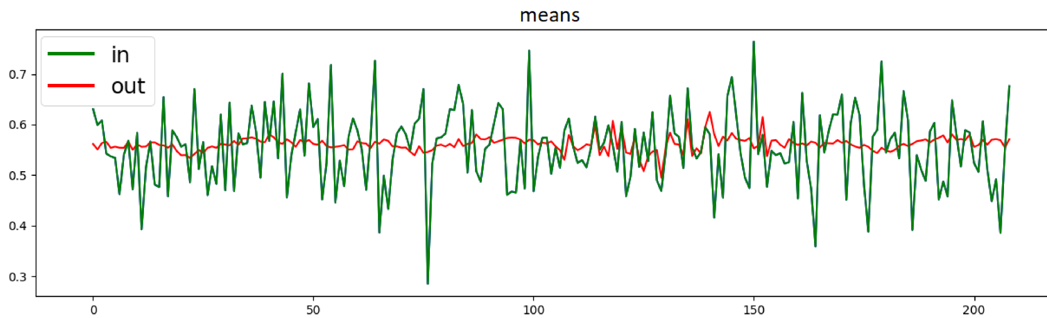
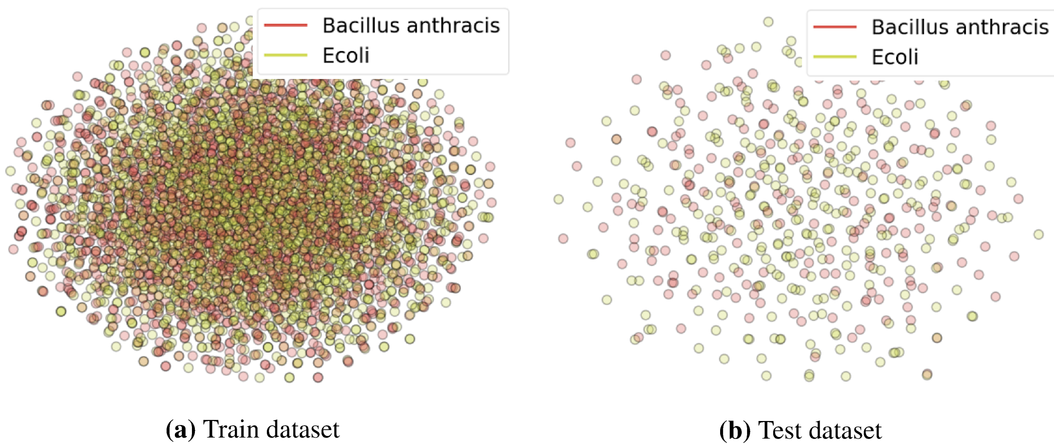


Figure 4.7: Variational Convolutional Autoencoder output on 2 classes



(a) Train dataset

(b) Test dataset

Figure 4.8: Variational Convolutional Autoencoder results on 2 classes, visualized with t-SNE

Table 4.7: Variational Convolutional Autoencoder experiment on 2 classes

Parameters	
epochs	1200
encoding size	42

(a) Parameters

Metrics	
accuracy	0.5902

(b) Metrics

Table 4.8: Variational Convolutional Autoencoder structure

Layer name	No. of filters	Kernel size	Activation
encoder			
Conv1D	16	3	ReLU
Conv1D	32	3	ReLU
Conv1D	64	7	ReLU
Linear (mean)	32		
Linea(stdev)	32		
decoder			
Conv Transpose 1D	32	7	ReLU
Conv Transpose 1D	64	3	ReLU
Conv Transpose 1D	4	3	ReLU
Linear	timesteps		Tanh

4 and 6 classes

Similar to the experiments with no success on 2 classes, experiments with 4 and 6 classes did not do any better.

Discussion

Some architectures manage to reconstruct the signal better than others, but no architecture was successful in clustering the encodings.

The undercomplete autoencoder (figure 4.3) did reconstruct a general outline of the signal, with some regions better covered than others. Where it failed is reconstructing peaks and extremes in the signal, keeping a rather flat trajectory. No clusters were separated, even when training with a larger encoding size.

The output of the denoising autoencoder in figure 4.5 is quite interesting, as two distinct regions are reconstructed very well while others are not reconstructed at all (the flat sections). This pattern of some regions being covered better was occurring again and again through the experiments. No clusters are found.

The variational autoencoder had the worst performance, with the reconstructed signal only occasionally following the target sequence. A guess as to why its performance is bad could be that the underlying distribution of the raw signal too complex to be approximated.

An obvious solution to all poor performing models could be to simply increase the encoding size or train a deeper model or train for a lot more epochs. These are all valid points, but the risk of overfitting is quite high - a powerful autoencoder could learn to replicate the signal perfectly, but would still fail to learn anything meaningful about the signal. The latent representation would then contain no information about the given sample.

4.1.2. LSTM Autoencoders

The architecture of an LSTM Autoencoder is shown in figure 4.9. LSTM encoder and decoder consist of some number of stacked LSTM layers (one or more). The signal encoding is the last hidden state of the LSTM encoder.

Before feeding into the decoder network, the signal encoding is repeated $timestep$ times and that vector is the input to the decoder. This means that for each timestep the decoder will get the same input (our signal encoding), but a different hidden state from the LSTM cell.

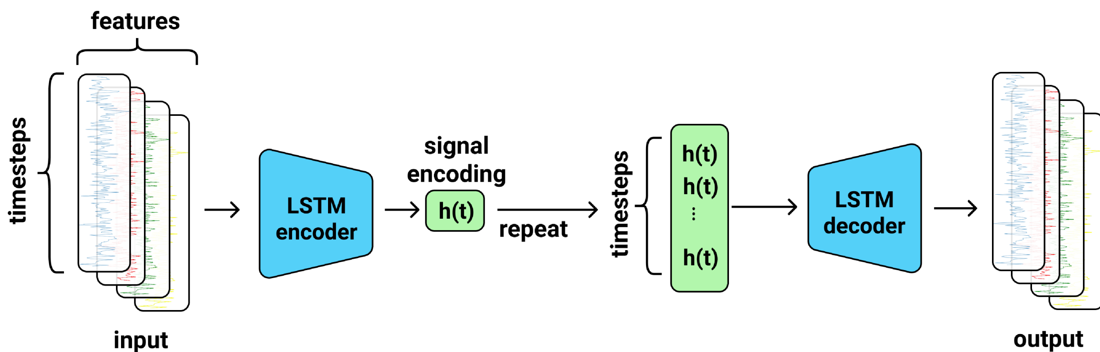


Figure 4.9: LSTM Autoencoder Structure: after the input is fed through the encoder, its last hidden state is repeated $timestep$ times and fed to the decoder

The process is analogous for a variational autoencoder with a slightly different bridge between the encoder and decoder, as shown before in section 3.11.

Undercomplete

2 classes

Results for this experiment are shown in figures 4.10 and 4.11, parameters in table 4.9a and metrics in table 4.9b.

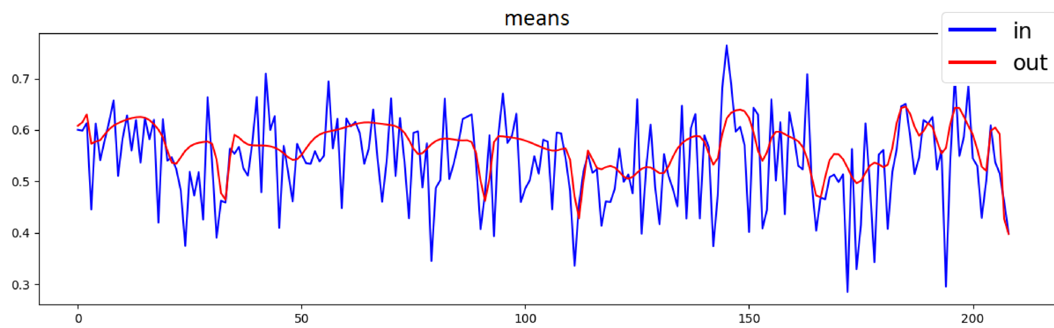
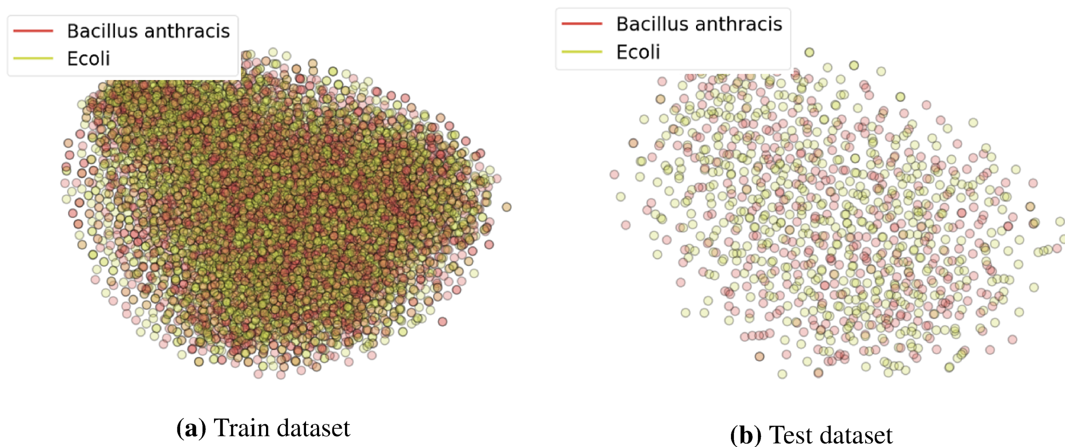


Figure 4.10: Undercomplete LSTM Autoencoder output on 2 classes



(a) Train dataset

(b) Test dataset

Figure 4.11: Undercomplete LSTM Autoencoder results on 2 classes, visualized with t-SNE

Table 4.9: Undercomplete LSTM Autoencoder experiment on 2 classes

Parameters	
epochs	3000
encoding size	64
layers	2

(a) Parameters

Metrics	
accuracy	0.5659

(b) Metrics

4 classes

Results for this experiment are shown in figures 4.12 and 4.13, parameters in table 4.10a and metrics in table 4.10b.

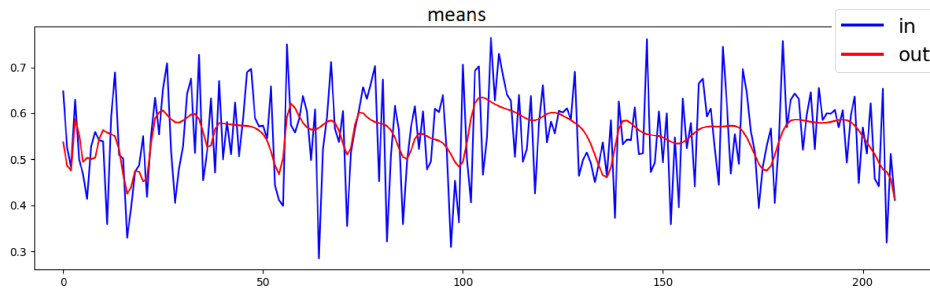
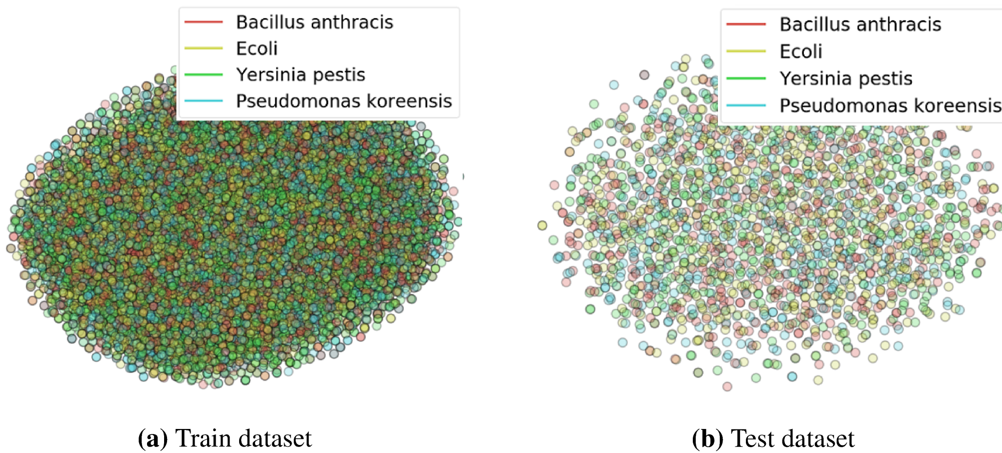


Figure 4.12: Undercomplete LSTM Autoencoder output on 4 classes



(a) Train dataset

(b) Test dataset

Figure 4.13: Undercomplete LSTM Autoencoder results on 4 classes, visualized with t-SNE

Table 4.10: Undercomplete LSTM Autoencoder experiment on 4 classes

Parameters	
epochs	1000
encoding size	128
layers	3

(a) Parameters

Metrics	
accuracy	0.3683

(b) Metrics

6 classes

Results for 6 classes were no significantly better than results on 2 and 4 classes.

Denoising Autoencoder

2 classes

Results for this experiment are shown in figures 4.14 and 4.15, parameters in table 4.11a and metrics in table 4.11b.

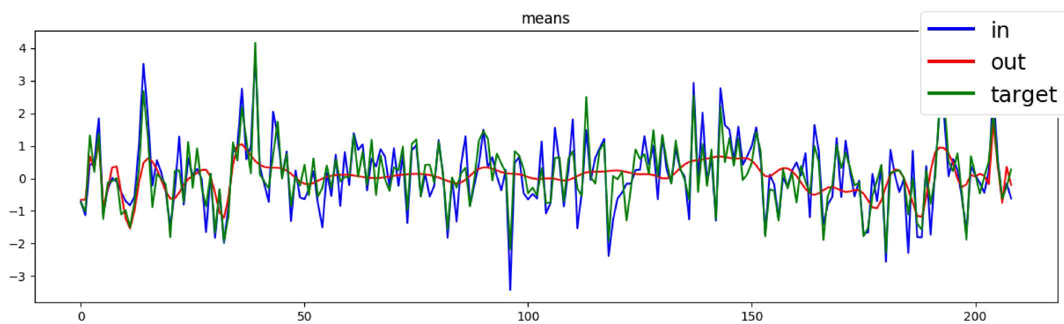
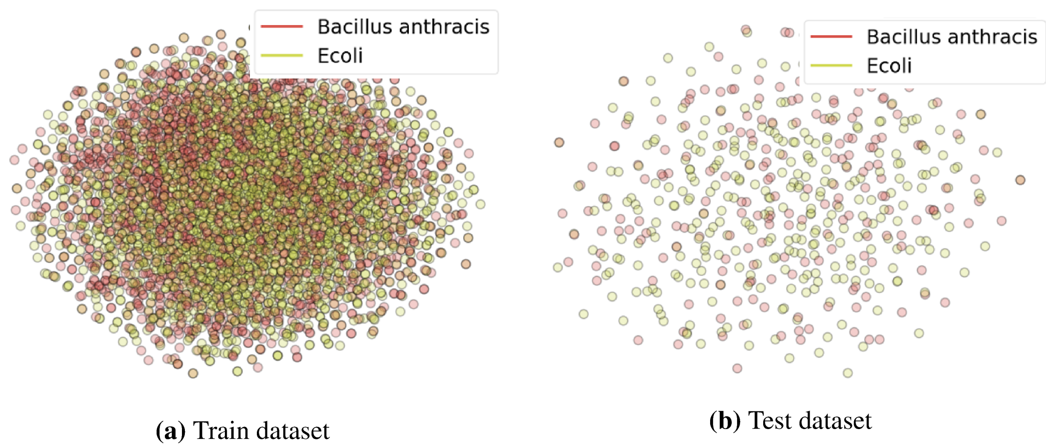


Figure 4.14: Denoising LSTM Autoencoder output on 2 classes



(a) Train dataset

(b) Test dataset

Figure 4.15: Denoising LSTM Autoencoder results on 2 classes, visualized with t-SNE

Table 4.11: Denoising LSTM Autoencoder experiment on 2 classes

Parameters	
epochs	2000
encoding size	100

(a) Parameters

Metrics	
accuracy	0.56

(b) Metrics

6 classes

Results for this experiment are shown in figures 4.16 and 4.17, parameters in table 4.12a and metrics in table 4.12b.

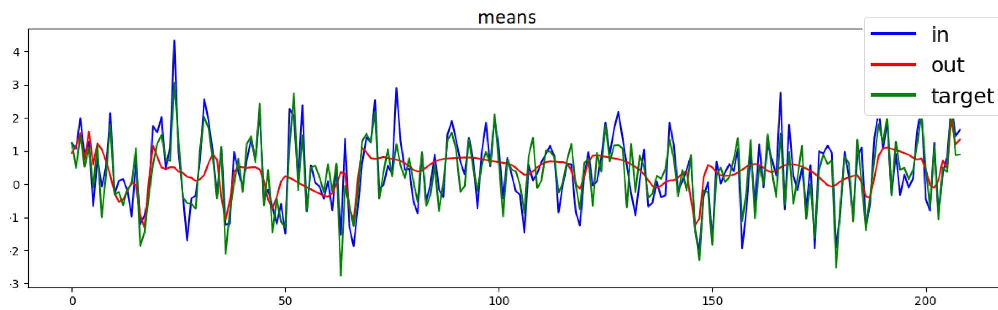
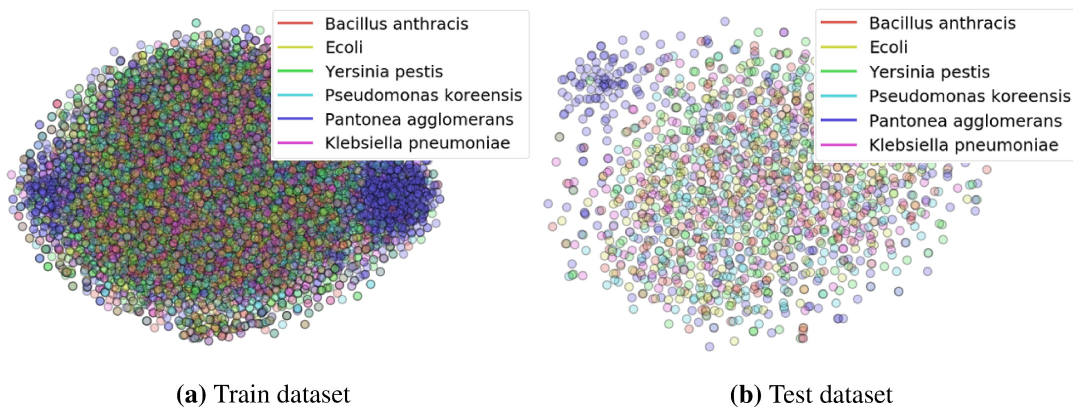


Figure 4.16: Denoising LSTM Autoencoder output on 6 classes



(a) Train dataset **(b)** Test dataset

Figure 4.17: Denoising LSTM Autoencoder results on 6 classes, visualized with t-SNE

Variational

No significant results here.

Table 4.12: Denoising LSTM Autoencoder experiment on 6 classes

Parameters	
epochs	1500
encoding size	200

(a) Parameters

Metrics	
accuracy	0.31

(b) Metrics

Discussion

LSTM Autoencoders performed similarly to Convolutional Autoencoders.

The Undercomplete LSTM Autoencoder was reconstructing a very general outline of the original signal. The reconstruction followed some peaks, but the model was trained for a while with no further improvements.

The output of the Denoising LSTM Autoencoder again reconstructs two regions better than the rest of the signal, which is interesting considering the architecture is different (in comparison to the Convolutional Autoencoder).

The Variational LSTM Autoencoder performed worse by far (thus no results included above), with the reconstruction being almost a flat line.

4.1.3. Autoencoder Performance Summary

A fact to remember when thinking about autoencoder results is that the reconstruction plots above show only $\frac{1}{4}$ of the data fed into the autoencoder. The plots show only the means signal, while the autoencoder was working on reconstructing medians, standard deviations, and interquartile ranges as well.

Another thing to think about is the reconstruction versus compression autoencoder usage. The goal of this work was not just to reconstruct the signal, but achieve generalized compression - i.e. to be able to generate a compressed encoding of an unseen sample. As seen in result images, the compressed encodings the autoencoders learned were not successful in separating clusters of data even though in some cases the autoencoder did loosely manage to reconstruct some parts of the input signal. However, learning encodings that cluster the data is not the actual job of the autoencoder as it is trained on pure reconstruction loss.

All in all, the autoencoder performance was not satisfactory.

4.2. Triplet Networks

Dataset

Dataset sizes can be seen in table 4.13.

Table 4.13: Triplet dataset sizes

	2 classes	4 classes	6 classes
Training set	16 758	100 548	253 180
Validation set	2 184	12 474	31 650
Test set	1 093	2 080	3 166

These datasets are larger in size than the ones in autoencoder networks (table 4.2) because they are formed of triplets, as shown in algorithm 1.

Triplets are formed in a way where for each input sample, n samples from a class different from the class of the sample are randomly chosen from the dataset. Those samples are used as the $negative$ part of the triplet, while the $positive$ samples are randomly chosen from the same class as the input sample.

Algorithm 1 Triplet formation

```
for sample, label in dataset do
    for other in classes do
        if label is not other then
            positive ← n samples of class label
            negative ← n samples of class other
            triplets ← (n ∗ sample, triplet positive, triplet negative)
            add triplet to triplet dataset
        end if
    end for
end for
```

Effectively, one input sample yields $n * (num\ classes - 1)$ samples, where n is the desired number of triplets formed from one input sample.

In these experiments, $n = 2$, so each input sample will yield 2 triplets per class different from the input sample class. The exception is the test set, for which only the anchor is used for evaluation, so no additional samples are generated ($n = 1$).

Metrics

Since triplet network results are much better than the autoencoder results, more metrics were calculated for these experiments.

The metrics include precision, recall, F1 score, and accuracy as usual.

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations:

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{4.1}$$

Recall (sensitivity) is the proportion of correctly classified positive observations to the all observations in the actual class:

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{4.2}$$

F1 score is the weighted average (otherwise named the harmonic mean) of precision and recall:

$$F1\ score = 2 * \frac{precision\ *\ recall}{precision + recall} \tag{4.3}$$

Accuracy is the proportion of correctly classified samples to all samples:

$$accuracy = \frac{true\ positive + true\ negative}{positive + negative} \tag{4.4}$$

In a multi-class scenario such as this one, the metrics are calculated for each class and then averaged (so-called macro averaging).

These metrics are more useful in scenarios of imbalanced classification - with one category representing the majority of data points, such as e.g. the rate of a rare disease in a population. Still, it can be convenient to have more information about the performance of a model.

4.2.1. Convolutional Triplet Network

2 classes

Network structure for this experiment is shown in table 4.18, results in figure 4.19, parameters in table 4.14a and metrics in table 4.14b.

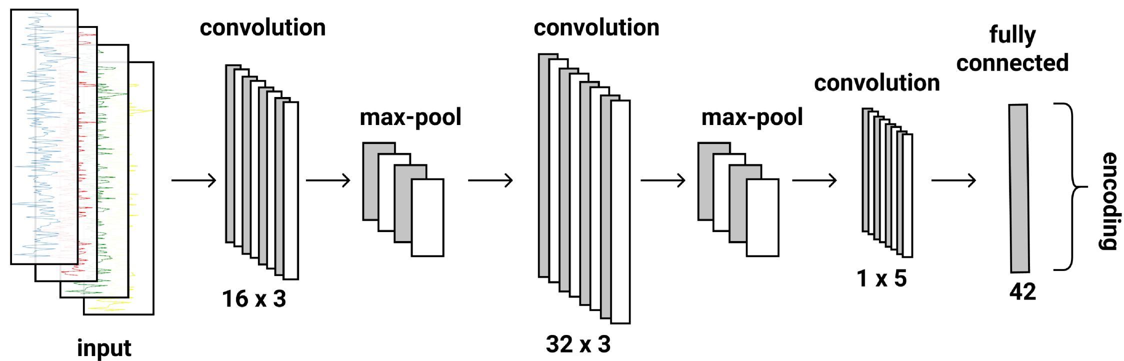
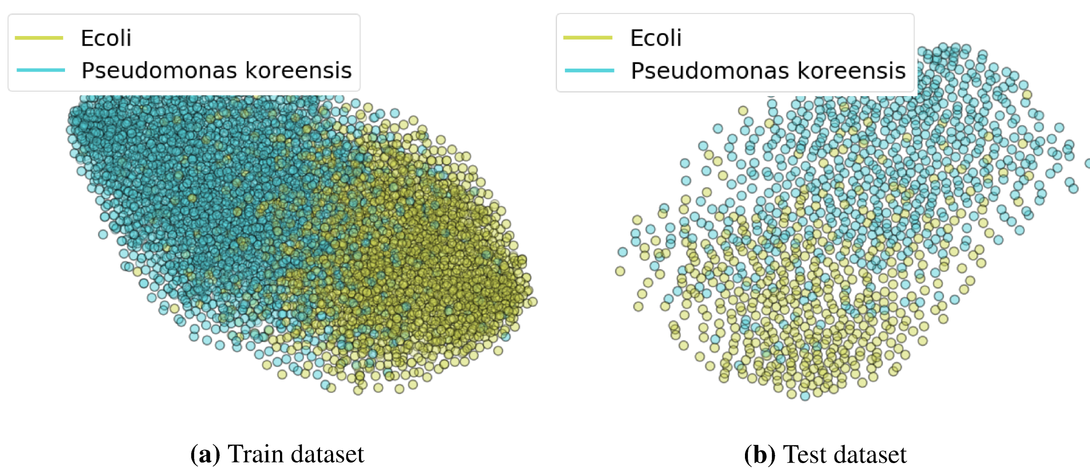


Figure 4.18: Convolutional Triplet Network Structure on 2 classes



(a) Train dataset **(b)** Test dataset

Figure 4.19: Convolutional Triplet Network results on 2 classes, visualized with t-SNE

4 classes

Network structure for this experiment is shown in figure 4.20, results in figure 4.21, parameters in table 4.15a and metrics in table 4.15b.

Table 4.14: Convolutional Triplet Network experiment on 2 classes

Parameters	
epochs	50
encoding size	42

(a) Parameters

Metrics			
precision	0.8263	recall	0.8177
F1	0.8208	accuracy	0.8265

(b) Metrics

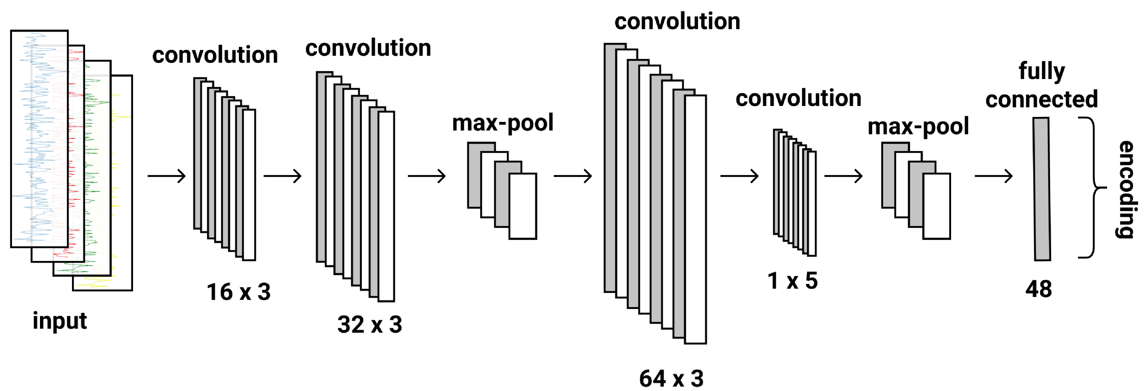
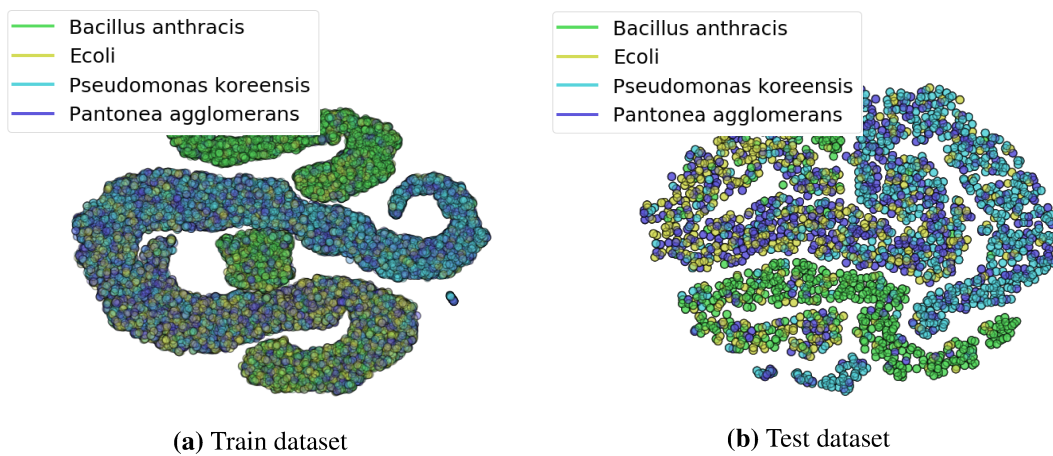


Figure 4.20: Convolutional Triplet Network Structure on 4 classes



(a) Train dataset

(b) Test dataset

Figure 4.21: Convolutional Triplet Network results on 4 classes, visualized with t-SNE

Table 4.15: Convolutional Triplet Network experiment on 4 classes

Parameters	
epochs	75
encoding size	48

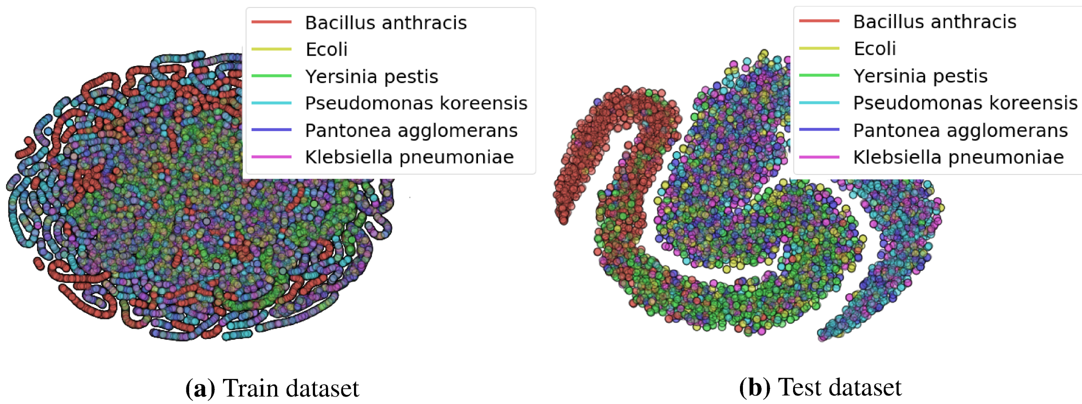
(a) Parameters

Metrics			
precision	0.5559	recall	0.5815
F1	0.5815	accuracy	0.5889

(b) Metrics

6 classes

Results for this experiment are shown in figure 4.22, network structure in figure 4.23, parameters in table 4.16a and metrics in table 4.16b.



(a) Train dataset

(b) Test dataset

Figure 4.22: Convolutional Triplet Network results on 6 classes, visualized with t-SNE

Table 4.16: Convolutional Triplet Network experiment on 6 classes

Parameters	
epochs	100
encoding size	50

(a) Parameters

Metrics			
precision	0.5392	recall	0.5298
F1	0.5276	accuracy	0.534

(b) Metrics

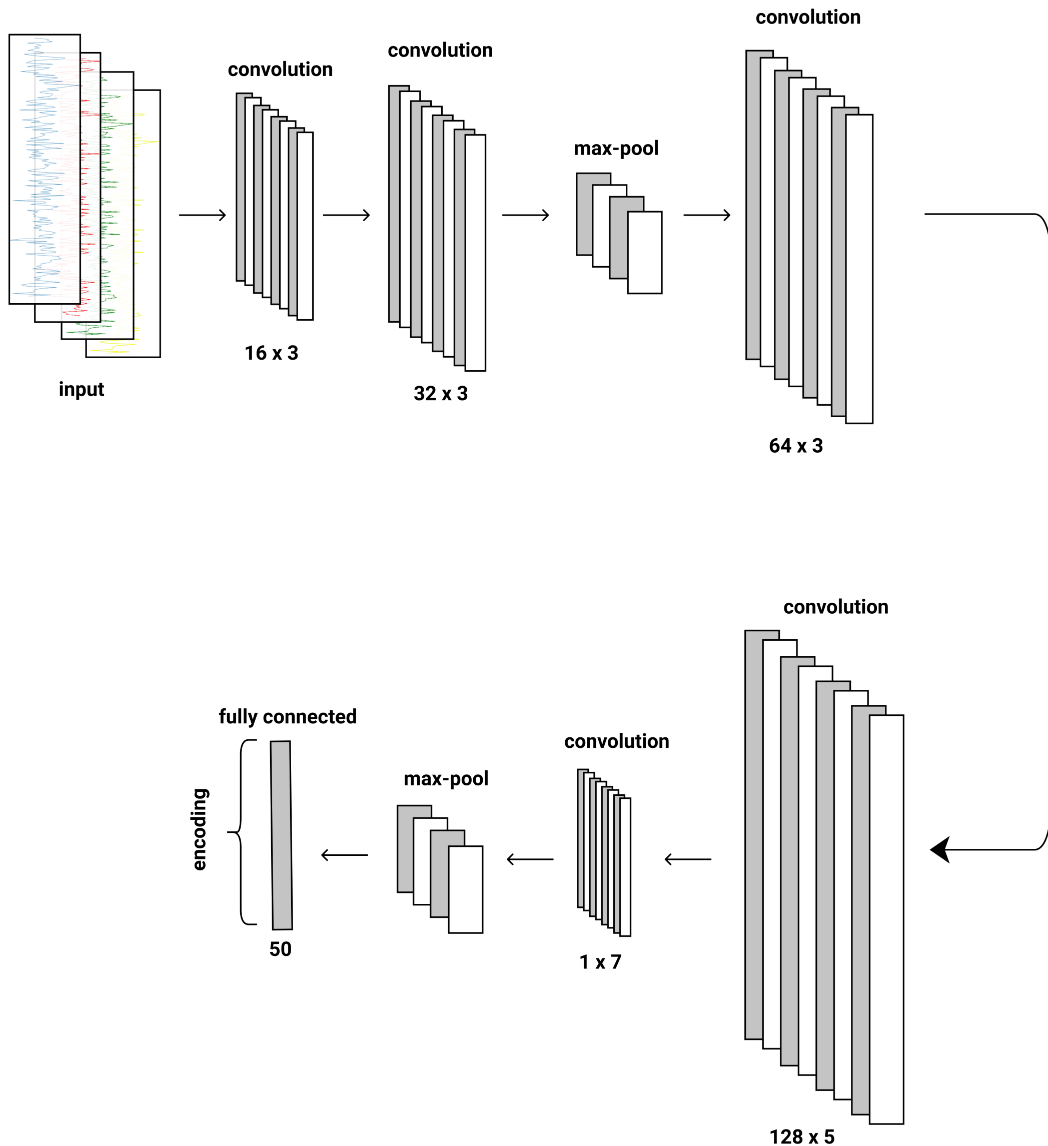


Figure 4.23: Convolutional Triplet Network Structure on 6 classes

Discussion

Convolutional Triplet Networks were kept relatively shallow, with few convolutional layers. Adding more layers was causing the network to quickly overfit on training data. The overfitting was a recurring problem in these experiments, even with methods like batch normalization which should help regularize the network.

The number of channels and filter sizes were lower at the start of the networks and increased gradually in order to extract more representative, high-level information. Starting with the shape of 4 input channels (mean, median, standard deviation, interquartile range), the data was in the end reduced to 1 channel which represents the

learned data encoding.

The convolutional triplet networks managed to detect some data clustering, much more successfully than autoencoders. The 2-class experiment was quite successful with a nice division between the classes. The 4-class experiment did separate some clusters more successfully than others, but there are still opportunities for further enhancements. The same goes for the 6-class experiment.

What is interesting to note is the snake-like shape of the visualized data, more visible in 4-class and 6-class experiments. This happens because the model tries to separate the points according to a fixed margin, and this shape is mathematically more optimal for a multi-class problem.

4.2.2. LSTM Triplet Network

Some parameters, shown in 4.17, were kept constant across training runs.

Table 4.17: LSTM Constant Parameters

Constants	
loss margin	0.2
bidirectional	True

The general structure of the network for all experiments can be seen in figure 4.24. The signal encoding in this scenario is the LSTM's hidden state at the last timestep.

What will differ between experiments is the number of stacked LSTM layers, the dropout rate between layers, and the number of hidden units.

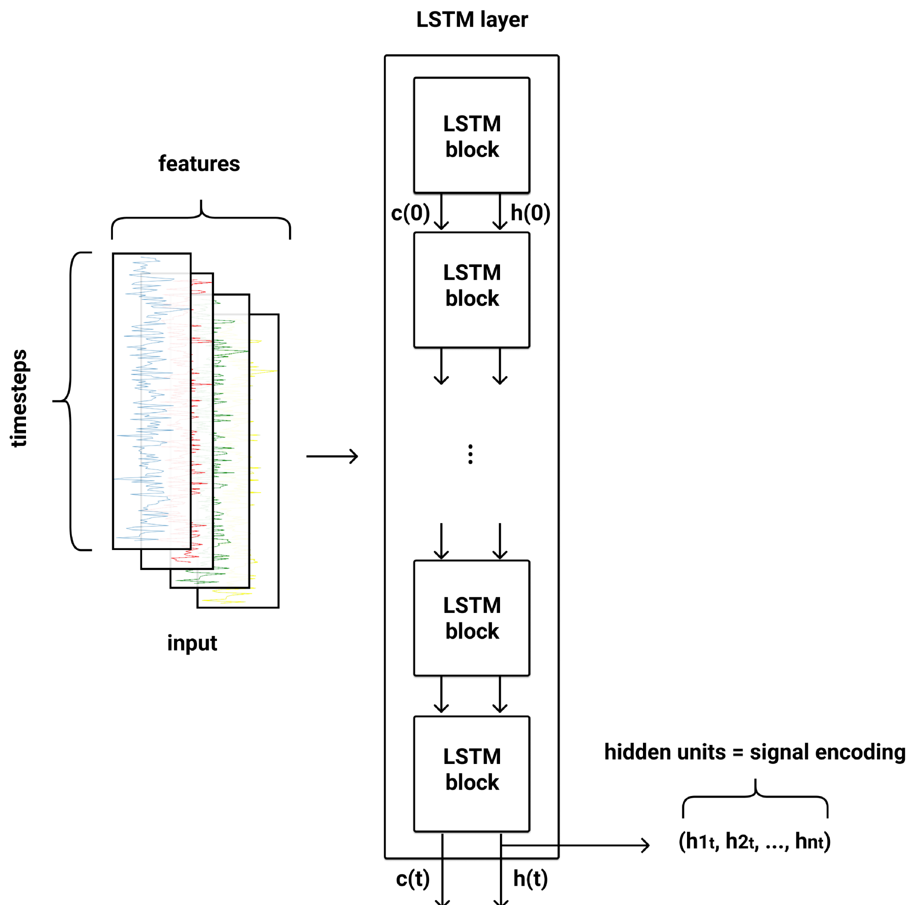
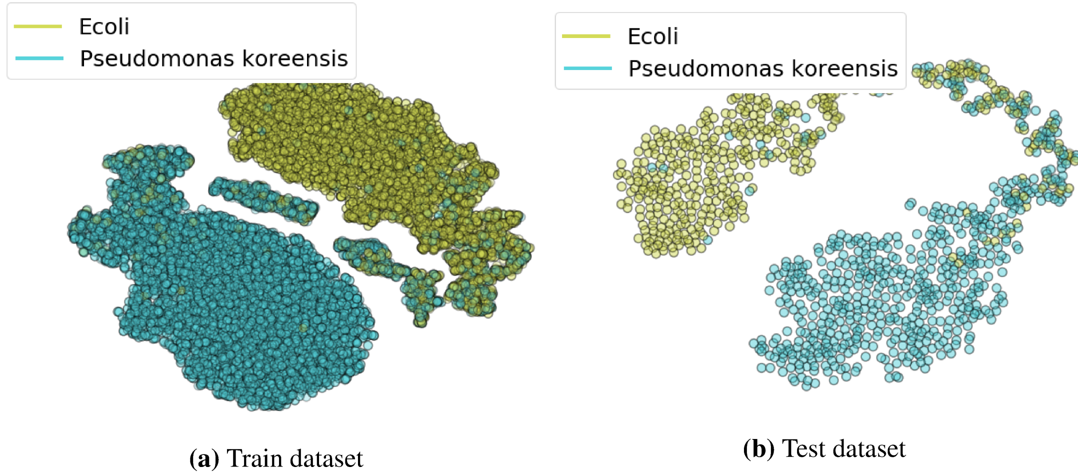


Figure 4.24: Triplet LSTM Network Structure: input is fed into a LSTM layer, its hidden state at the last timestep is the signal encoding

2 classes

Results for this experiment are shown in figure 4.25, parameters in table 4.18a and metrics in table 4.18b.



(a) Train dataset

(b) Test dataset

Figure 4.25: Triplet LSTM Network results on 2 classes, visualized with t-SNE

Table 4.18: LSTM Triplet Network experiment on 2 classes

Parameters			
epochs	50	encoding size	24
layers	2	dropout	0.3

(a) Parameters

Metrics			
precision	0.9603	recall	0.9088
F1	0.9344	accuracy	0.9224

(b) Metrics

4 classes

Results for this experiment are shown in figure 4.26, parameters in table 4.19a and metrics in table 4.19b.

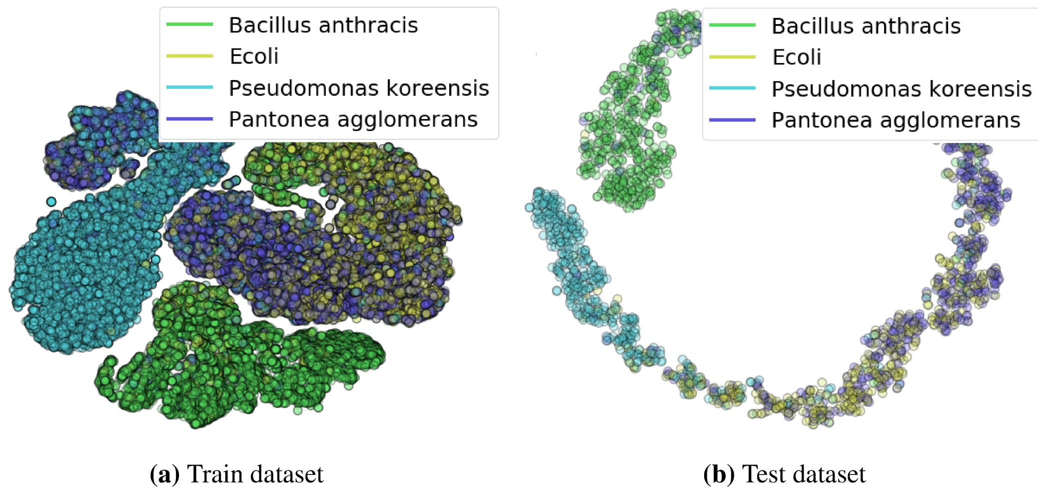
Table 4.19: LSTM Triplet Network experiment on 4 classes

Parameters			
epochs	100	encoding size	64
layers	2	dropout	0.5

(a) Parameters

Metrics			
precision	0.7396	recall	0.742
F1	0.7405	accuracy	0.7596

(b) Metrics

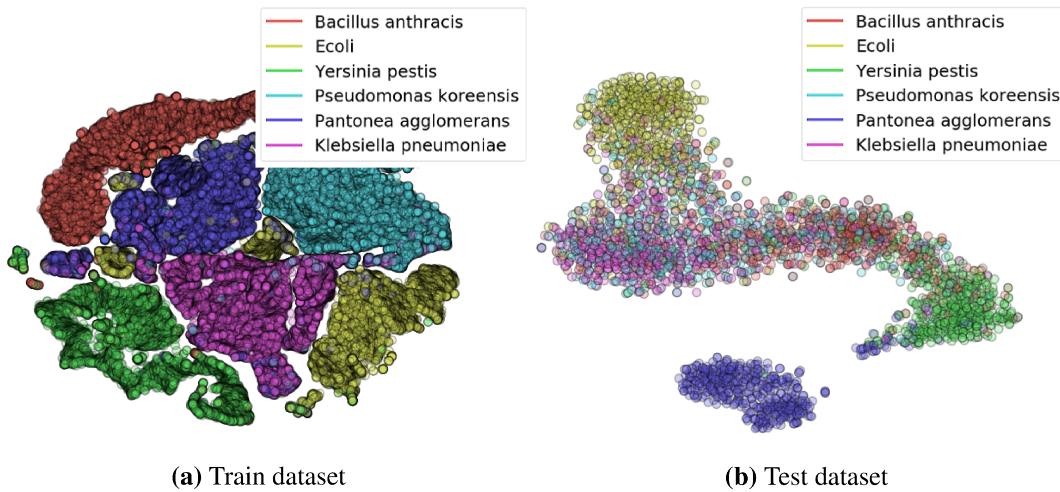


(a) Train dataset **(b)** Test dataset

Figure 4.26: Triplet LSTM Network results on 4 classes, visualized with t-SNE

6 classes

Results for this experiment are shown in figure 4.27, parameters in table 4.20a and metrics in table 4.20b.



(a) Train dataset **(b)** Test dataset

Figure 4.27: Triplet LSTM Network results on 6 classes, visualized with t-SNE

Table 4.20: LSTM Triplet Network experiment on 6 classes

Parameters			
epochs	200	encoding size	128
layers	2	dropout	0.5

(a) Parameters

Metrics			
precision	0.6541	recall	0.666
F1	0.6551	accuracy	0.683

(b) Metrics

Discussion

These experiments were by far most successful.

On the 2-class experiment the clusters are very distinctly separated. The encoding size is quite low which is good - only 24 units, whereas the dimensionality of input data is ~ 1000.

The 4-class experiment shows two more or less finely divided clusters with two classes still somewhat mixed together. This is still a valid result considering the dimensionality of the problem.

Similar results on the 6-class experiment - 3 distinct clusters with 3 classes still mixed up in the test dataset.

4.2.3. Further Experiments

Since the LSTM Triplet Network had good results, further experiments were made in an attempt to enhance its performance even more. All these experiments were done on a 6 class scenario.

Signal Energy

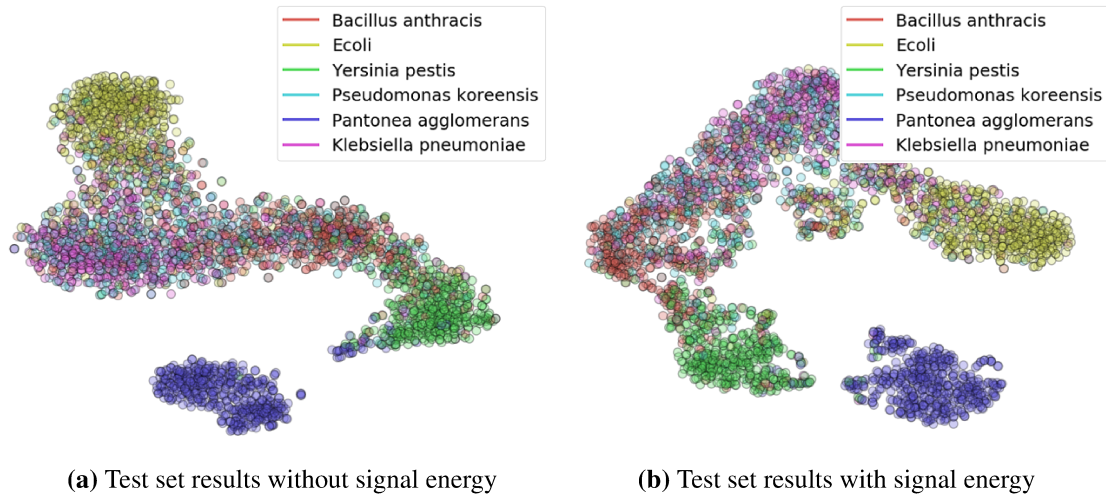
To describe the original signal better, an additional data feature was calculated during preprocessing. This feature was **signal energy**, which for a discrete-time signal is defined as

$$E_s = \sum_{n=-\infty}^{\infty} |x(n)|^2 \tag{4.5}$$

By using this additional feature, the data is now in shape $(timesteps/400, 5)$.

To get as comparable results as possible, parameters for the experiment with added signal energy were kept the same as the experiment with no added energy - as in table 4.20a.

The results of this experiment are shown in figure 4.28. Subfigure 4.28a is the same as subfigure 4.27a. The comparison of results is shown in table 4.21. Parameters were as in table 4.20a.



(a) Test set results without signal energy

(b) Test set results with signal energy

Figure 4.28: LSTM Triplet Network experiment results on 6 classes with and without added signal energy, visualized with t-SNE

Table 4.21: LSTM Triplet Network experiment metrics on 6 classes, with and without signal energy

Metric	Without Energy	With Energy
precision	0.6541	0.6534
recall	0.666	0.6706
F1	0.6551	0.6571
accuracy	0.683	0.6672

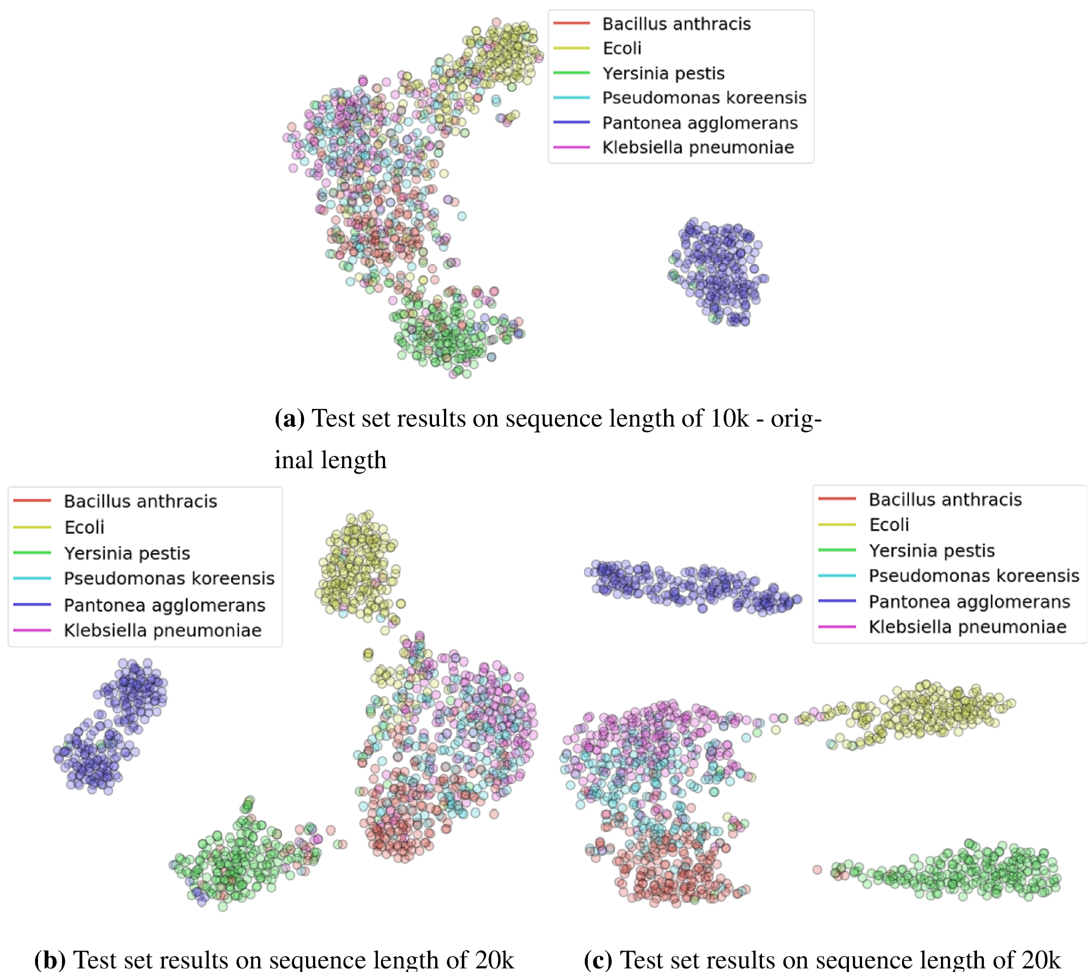
As can be seen both from result images and metrics, adding signal energy did not significantly improve model performance. However, a good plan for further research would be to try and find an optimal combination of statistical parameters in order to describe the signals as closely as possible.

In this work, only generic statistical coefficients (mean, median, standard deviation, and interquartile range) were used, but it is a good idea to include some signal-specific measures such as the signal energy used in this experiment.

Longer Sequences

The original dataset, described in section 2, was preprocessed by splitting reference genomes into smaller sections. The length of the originally made sections was 10k nucleotides. Further experiments were conducted where the genomes were split into sections of 20k and 50k nucleotides, to see if starting sequence length has any impact on the results. This experiment was done on a smaller scale, with 2000 examples from each class, in interest of saving time and resources.

Results are shown in figure 4.29, parameters in table 4.22a, and comparison of metrics in table 4.22b. Parameters were kept constant through experiments so that the model performance could be assessed objectively.



(a) Test set results on sequence length of 10k - original length

(b) Test set results on sequence length of 20k

(c) Test set results on sequence length of 20k

Figure 4.29: Test set results on experiment with longer sequences, visualized with t-SNE

Even though not as visible in the resulting images, longer sequences did improve model performance significantly, best seen from table 4.22b - accuracy improved from 65.83% in a 10k scenario to 83.75% in a 50k scenario.

Table 4.22: LSTM Triplet Network experiment results on 6 classes, with different starting sequence length

Parameters	
epochs	75
encoding size	64

(a) Parameters

Metric	10k	20k	50k
precision	0.6168	0.7477	0.8322
recall	0.6336	0.7375	0.8309
F1	0.6199	0.7397	0.8306
accuracy	0.6583	0.7417	0.8375

(b) Metric comparison

By refactoring from 10k to 20k sequence length, model accuracy improved by almost 10%, with more prominent clustering in the resulting images. Refactoring from 20k to 50k did not improve accuracy proportionally to the increased length, with the accuracy again improving by a little less than 10%. Resulting clusters were comparably good as well.

The reason for model improvement when trained on longer sequences is simply that the model has more information about the sequence itself. Moreover, since the sliding window offset is kept constant at 1000 bp and the window length is increased, the sequence overlap increases as well. The model then has more context for each preprocessed sequence and how it relates to other sequences.

What needs to be emphasized here is that prolonging the sequences is feasible only in the artificial dataset preprocessing. This step was done in order to simulate the actual reads - when the model is trained on real data one has no influence on read lengths.

Time and Memory Usage

An interesting comparison of time and memory usages can be done along with this experiment. Table 4.23 shows total training time for 75 epochs, and total size of the dataset in memory for different sequence lengths.

One should note that this is a dataset of reduced size in comparison to other experiments.

Table 4.23: Comparison of time and memory usage

Metric	10k	20k	50k
total training time	54m	1h 47m	3h 15m
total dataset size	128.5 MB	256.2 MB	641.1 MB

As expected, the training time and memory usage increase proportionally with the increase of sequence lengths. Increasing sequence length will improve performance, but with the cost of more resource usage - there needs to be a compromise in order to get best performance while still being able to work with the dataset.

Another thing to note here is that the dataset consists of only 6 microbial species, whereas an actual, real-life scenario would include tens of thousands of different microbes.

4.2.4. Triplet Networks Performance Summary

Triplet networks in general were much more successful than autoencoders. This is only logical since triplet networks learn directly on signal encodings - where autoencoders tried to simply reconstruct a given input, triplet networks worked on actually separating the encodings of different classes.

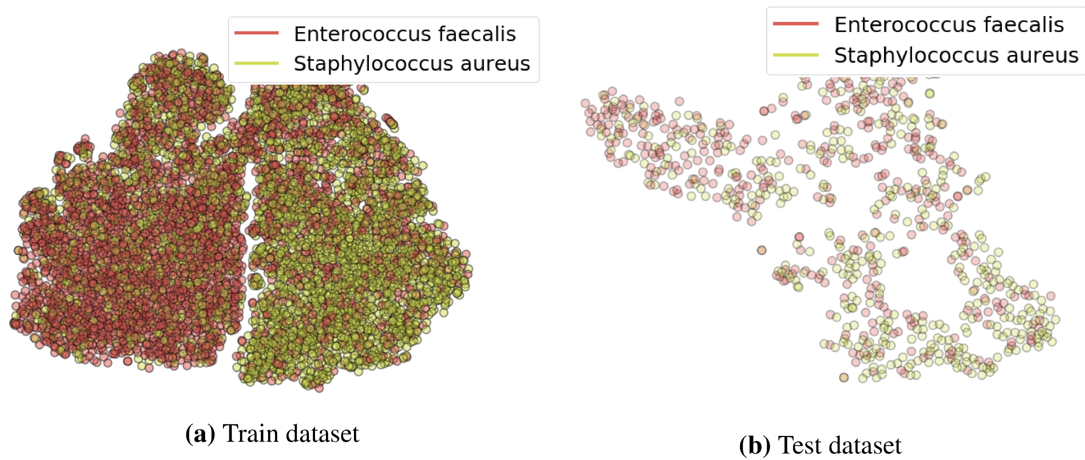
In terms of the comparison of convolutional and LSTM architectures, the results are as expected - LSTM networks performed better. LSTMs were designed for problems like these - long sequences with important data potentially multiple timesteps apart. This is why LSTMs were usually applied to sequence modelling problems, while convolutional networks work better with images where the entirety of an input sample is from a fixed point in time.

5. Evaluation

5.1. Results on the Zymo mock community dataset

2 classes

Results for this experiment are shown in figure 5.1, parameters in table 5.1a and metrics in table 5.1b.



(a) Train dataset

(b) Test dataset

Figure 5.1: Zymo dataset Triplet LSTM Network results on 2 classes, visualized with t-SNE

Table 5.1: Zymo dataset 2 class Triplet LSTM Network experiment

Parameters			
epochs	1000	encoding size	48
layers	1	dropout	0.1

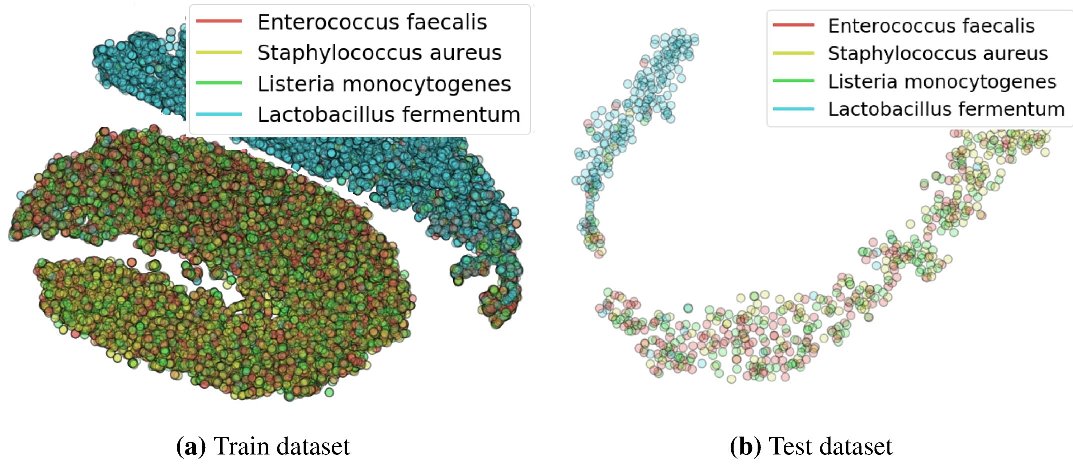
(a) Parameters

Metrics			
precision	0.67	recall	0.6658
F1	0.6612	accuracy	0.6625

(b) Metrics

4 classes

Results for this experiment are shown in figure 5.2, parameters in table 5.2a and metrics in table 5.2b.



(a) Train dataset

(b) Test dataset

Figure 5.2: Zymo dataset Triplet LSTM Network results on 4 classes, visualized with t-SNE

Table 5.2: Zymo dataset 4 class Triplet LSTM Network experiment

Parameters			
epochs	200	encoding size	64
layers	3	dropout	0.25

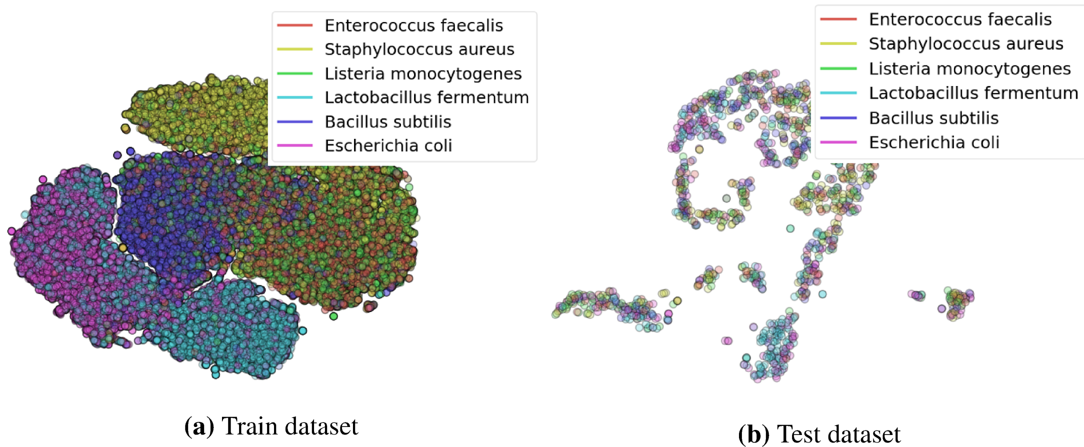
(a) Parameters

Metrics			
precision	0.5659	recall	0.5781
F1	0.5691	accuracy	0.5625

(b) Metrics

6 classes

Results for this experiment are shown in figure 5.3, parameters in table 5.3a and metrics in table 5.3b.



(a) Train dataset

(b) Test dataset

Figure 5.3: Zymo dataset Triplet LSTM Network results on 6 classes, visualized with t-SNE

Table 5.3: Zymo dataset 6 class Triplet LSTM Network experiment

Parameters			
epochs	500	encoding size	128
layers	3	dropout	0.5

(a) Parameters

Metrics			
precision	0.3058	recall	0.3179
F1	0.3027	accuracy	0.3333

(b) Metrics

5.2. Discussion

Results on the Zymo mock dataset are worse than those from the artificial dataset. This is expected as these signals are true signals - not artificially generated. They are also more diverse in length (some being much shorter than the others) which means that the preprocessing pipeline could be improved on these signals, perhaps by using more statistical coefficients to describe the data. What is more, there is the possible additional introduced error through the Kraken and cross-referencing pipeline.

These results are also from a small fraction of the entire sequenced dataset, which can easily mean that some meaningful features are lost in those omitted signals.

6. Conclusion

The main focus of this work is on nanopore raw signals. A raw signal in nanopore sequencing is generated when a DNA strand is driven through a nano-scale hole, where each nucleotide induces a change in current when passing through. The generated signals are not even close to perfect, due to the background noise of the sequencing technology and the pure nature of the problem. The goal is to extract relevant features from these signals so that one could differ between microbial species based on that information.

After preprocessing the dataset, this work tests multiple types of models in hope of finding the best compressed signal representations which would contain relevant information about the given signal. After training a model, its success is tested by using a classifier on a test dataset of unseen samples. The models include autoencoders (undercomplete, denoising, and variational) and triplet networks, both using convolutional and LSTM architectures.

Of the architectures specified above, LSTM triplet networks achieved the best results and were most successful in identifying clusters of embedded representations. The LSTM triplet architecture was therefore evaluated further on actual signal data from the Zymo mock community publicly available dataset.

Future work

As far as future work goes, autoencoders can be researched further to see if the problem was in the architecture itself or the dataset is simply too complicated for the autoencoder to manage.

Another improvement could be to improve the loss function of the autoencoder - instead of trying to simply duplicate the signal, an additional term could be added which would regulate how close the actual signal encodings of the same class are to each other. This would potentially nudge the autoencoder into learning features specific to each microbe.

If an autoencoder gives solid signal reconstruction but separates the compressed signals poorly, another idea is to try to separate autoencoder encodings further using triplet loss - essentially combining the two network architectural concepts used in this work.

As already mentioned, a possible improvement on both autoencoders and triplet networks is a different combination of statistical coefficients used to describe the signals. Instead of using generic descriptive coefficients, signal-specific measures such as signal energy, amplitude or even shifting the signal to the frequency domain could be useful to research.

BIBLIOGRAPHY

Guillaume Alain and Yoshua Bengio. What regularized auto-encoders learn from the data generating distribution. 2012.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning (Adaptive Computation and Machine Learning*. MIT Press, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

Elad Hoffer and Nir Ailon. Deep metric learning using triplet network, 2014.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

Imchang Lee, Yeong Ouk Kim, Sang-Cheol Park, and Jongsik Chun. Orthoani: An improved algorithm and software for calculating average nucleotide identity. *International Journal of Systematic and Evolutionary Microbiology*, 66(2):1100–1103, 2016. ISSN 1466-5026.

Yu Li, Renmin Han, Chongwei Bi, Mo Li, Sheng Wang, and Xin Gao. Deepsimulator: a deep simulator for nanopore sequencing. *Bioinformatics*, 34(17):2899–2908, 04 2018. ISSN 1367-4803.

Nicholas J. Loman, Samuel M. Nicholls, Joshua C. Quick, and Shuiquan Tang. Ultra-deep, long-read nanopore sequencing of mock microbial community standards. *bioRxiv*, 2018.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015.

M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

Laurens Van der Maaten and Geoffrey Hinton. Viualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 11 2008.

Derrick Wood and Steven Salzberg. Kraken: Ultrafast metagenomic sequence classification using exact alignment. *Genome biology*, 15, 2014.

Derrick Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with kraken 2. *Genome Biology*, 20, 2019.

LIST OF FIGURES

2.1. Sliding window concept with window length 25 and offset 10 5
2.2. Example of the first 1000 timesteps of a signal generated by DeepSimulator 5
2.3. Raw signal example 7
2.4. Preprocessed signal example. A raw signal is first split into fragments of 400 timesteps which are then each mapped to 4 statistics - mean, median, standard deviation, interquartile range. 7

3.1. Scheme of the work pipeline. Raw signals go through a preprocessing stage before they are fed into a model. The model learns signal representations, which are visualized with tSNE and classified to a microbial species. 10
3.2. Overview of models used in this work 11
3.3. Simple Artificial Neural Network architecture with an input layer, one hidden layer and an output layer 12
3.4. Effect of applying the sigmoid activation function multiple times . . . 13
3.5. Cross-correlation (convolution) applied to a 1D grid 14
3.6. Recurrent Neural Network architecture 15
3.7. LSTM cell scheme 16
3.8. Autoencoder architecture, consisting of an encoder which learns compressed representations, and a decoder which learns to map the representation back to the original output 17
3.9. Hidden variable z generating an observation x 19
3.10. Using known observations x to estimate the hidden variable z 20
3.11. Variational Autoencoder architecture: $p(x|z)$ is approximated with a distribution $q(z|x)$ 20

3.12. Variational Autoencoder implementation: encoder maps the input into μ and σ in latent space. A similar point z is sampled from $\mathcal{N}(\mu, \sigma)$, and fed to the decoder 21
3.13. The effect of the reparametrization trick on random sampling: introducing a new parameter ϵ randomly sampled from a unit Gaussian enables random sampling of z 22
3.14. Triplet learning: positive samples are pushed closer to the anchor and negative samples further from it 23
3.15. K-nearest neighbors algorithm idea in 2D feature space: the class of q is determined by the classes of k nearest neighbors 24

4.1. Test set representations before training, visualized with t-SNE 27
4.2. Input sequence with added Gaussian noise and target sequence 29
4.3. Undercomplete Convolutional Autoencoder output on 2 classes 30
4.4. Undercomplete Convolutional Autoencoder results on 2 classes, visualized with t-SNE 30
4.5. Denoising Convolutional Autoencoder output on 4 classes 31
4.6. Denoising Convolutional Autoencoder results on 4 classes, visualized with t-SNE 32
4.7. Variational Convolutional Autoencoder output on 2 classes 33
4.8. Variational Convolutional Autoencoder results on 2 classes, visualized with t-SNE 33
4.9. LSTM Autoencoder Structure: after the input is fed through the encoder, its last hidden state is repeated $timestep$ times and fed to the decoder 35
4.10. Undercomplete LSTM Autoencoder output on 2 classes 36
4.11. Undercomplete LSTM Autoencoder results on 2 classes, visualized with t-SNE 36
4.12. Undercomplete LSTM Autoencoder output on 4 classes 37
4.13. Undercomplete LSTM Autoencoder results on 4 classes, visualized with t-SNE 37
4.14. Denoising LSTM Autoencoder output on 2 classes 38
4.15. Denoising LSTM Autoencoder results on 2 classes, visualized with t-SNE 38
4.16. Denoising LSTM Autoencoder output on 6 classes 39

4.17. Denoising LSTM Autoencoder results on 6 classes, visualized with t-SNE 39
4.18. Convolutional Triplet Network Structure on 2 classes 43
4.19. Convolutional Triplet Network results on 2 classes, visualized with t-SNE 43
4.20. Convolutional Triplet Network Structure on 4 classes 44
4.21. Convolutional Triplet Network results on 4 classes, visualized with t-SNE 44
4.22. Convolutional Triplet Network results on 6 classes, visualized with t-SNE 45
4.23. Convolutional Triplet Network Structure on 6 classes 46
4.24. Triplet LSTM Network Structure: input is fed into a LSTM layer, its hidden state at the last timestep is the signal encoding 48
4.25. Triplet LSTM Network results on 2 classes, visualized with t-SNE . . 49
4.26. Triplet LSTM Network results on 4 classes, visualized with t-SNE . . 50
4.27. Triplet LSTM Network results on 6 classes, visualized with t-SNE . . 50
4.28. LSTM Triplet Network experiment results on 6 classes with and without added signal energy, visualized with t-SNE 52
4.29. Test set results on experiment with longer sequences, visualized with t-SNE 53

5.1. Zymo dataset Triplet LSTM Network results on 2 classes, visualized with t-SNE 56
5.2. Zymo dataset Triplet LSTM Network results on 4 classes, visualized with t-SNE 57
5.3. Zymo dataset Triplet LSTM Network results on 6 classes, visualized with t-SNE 58

List of Tables

2.1. Original Genome Lengths 3
2.2. Average Nucleotide Identity scores 4
2.3. Sample distribution through classes 8

4.1. Constant Parameters 28
4.2. Autoencoder dataset sizes 29
4.3. Undercomplete Convolutional Autoencoder experiment on 2 classes . 31
4.4. Undercomplete Convolutional Autoencoder structure 31
4.5. Denoising Convolutional Autoencoder experiment on 4 classes, visualized with t-SNE 31
4.6. Denoising Convolutional Autoencoder structure 32
4.7. Variational Convolutional Autoencoder experiment on 2 classes . . . 33
4.8. Variational Convolutional Autoencoder structure 34
4.9. Undercomplete LSTM Autoencoder experiment on 2 classes 36
4.10. Undercomplete LSTM Autoencoder experiment on 4 classes 37
4.11. Denoising LSTM Autoencoder experiment on 2 classes 39
4.12. Denoising LSTM Autoencoder experiment on 6 classes 40
4.13. Triplet dataset sizes 41
4.14. Convolutional Triplet Network experiment on 2 classes 44
4.15. Convolutional Triplet Network experiment on 4 classes 45
4.16. Convolutional Triplet Network experiment on 6 classes 45
4.17. LSTM Constant Parameters 48
4.18. LSTM Triplet Network experiment on 2 classes 49
4.19. LSTM Triplet Network experiment on 4 classes 49
4.20. LSTM Triplet Network experiment on 6 classes 51
4.21. LSTM Triplet Network experiment metrics on 6 classes, with and without signal energy 52

4.22. LSTM Triplet Network experiment results on 6 classes, with different starting sequence length 54
4.23. Comparison of time and memory usage 54

5.1. Zymo dataset 2 class Triplet LSTM Network experiment 56
5.2. Zymo dataset 4 class Triplet LSTM Network experiment 57
5.3. Zymo dataset 6 class Triplet LSTM Network experiment 58

Microbe Detection Using Deep Learning

Abstract

Microbes, omnipresent microorganisms invisible to the naked eye, impact many functions in the human body. The ability to detect and classify them is essential in order to discover diseases, prescribe medication, and keep a healthy lifestyle. The goal of this thesis is to develop a method for microbe detection based on a deep learning architecture. The architecture is designed to find suitable representations of signals corresponding to sequenced microbe DNA fragments. After finding the signal representations, an appropriate distance metric is used to separate different species in the latent space. In the end, reads are classified using a suitable classifier.

Keywords: bioinformatics, deep learning, triplet loss, autoencoder, classification

Prepoznavanje mikroba uporabom dubokog učenja

Sažetak

Mikrobi, sveprisutni mikroorganizmi nevidljivi golom oku, utječu na mnogo funkcija u ljudskom tijelu. Mogućnost njihovog prepoznavanja i klasificiranja je važna kod otkrivanja bolesti, prepisivanja lijekova i održavanja zdravog načina života. Cilj ovog rada je razvoj metode za detekciju mikroba koristeći metode dubokog učenja. Razvijena metoda pronalazi odgovarajuće reprezentacije signala očitanih za dani DNA fragment mikroba. Nakon pronalaska reprezentacija signala, odgovarajuća metrika je korištena za razdvajanje različitih vrsta u latentnom prostoru. U konačnici, očitanja su klasificirana koristeći prikladni klasifikacijski model.

Ključne riječi: bioinformatika, duboko učenje, trojni gubitak, autoenkoder, klasifikacija