Species Classification from Short Genomic Reads using Feedforward Neural Networks

submitted by EMRE ÖZZEYBEK in partial fulfillment of the requirements for the degree of Master of Science in Health Informatics, Middle East Technical University by,

Prof. Dr. Banu Günel Küçük
Dean, Graduate School of Informatics

Assoc. Prof. Dr. Yeşim Aydıñoğlu Son
Head of Department, Health Informatics

Assist. Prof. Dr. Aybar Can Acar
Supervisor, Health Informatics

Examining Committee Members:

Assoc. Prof. Dr. Yeşim Aydınoğlu
Health Informatics, METU

Assist. Prof. Dr. Aybar Can Acar
Health Informatics, METU

Assoc. Prof. Dr. Tunca Doğan
Computer Engineering, Hacettepe University

Date: 11.09.2023
I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Emre Özzeybek

Signature: 
With the cost of Next Generation Sequencing technologies in decline, the need for fast and efficient classification of genomic findings has become of utmost importance. Due to the output length limitations of most Second Generation Sequencing techniques, it is important that we are able to classify short reads of DNA. In this research, we trained a basic Artificial Neural Network model with three hidden layers on short reads(50-500 bp) taken from two species’ reference genomes. We selected Escherichia Coli and Saccharomyces Cerevisiae for their short and well-studied reference genomes. Their taxonomic difference makes them ideal candidates for ascertaining the viability of using the whole genome for species classification. We then classified these short reads. We achieved moderate success with a classification accuracy of 80% — 91% corresponding to differing hyperparameters and read lengths. We documented the encountered issues and considered future directions.

Keywords: Bioinformatics, Classification, Machine Learning, Neural Networks
ÖZ

BESLEMELİ SİNİR AĞLARI KULLANARAK GENOMİK KISA OKUMALARDAN TÜR SİNIFLANDIRMASI

Özzyebek, Emre
Yüksek Lisans, Biyoenformatik Programı
Tez Yöneticisi: Dr. Öğr. Üyesi. Aybar Can Acar

Eylül 2023, 45 sayfa


Anahtar Kelimeler: Biyoenformatik, Sınıflandırma, Makine Öğrenmesi, Sinir Ağıları
To my beloved family and friends...
Aileme ve dostlarım...
ACKNOWLEDGMENTS

Assist. Prof. Dr. Aybar Can Acar for all their support throughout my thesis and master’s program.

Assoc. Prof. Dr. Yeşim Aydın Son for their guidance.

Assoc. Prof. Dr. Tunca Doğan for their invaluable feedback.

Prof. Dr. Tolga Can for their enthusiastic lectures that introduced and lead me to Bioinformatics.

This thesis would not have come together if not for the unyielding support and constant push I got from my friends and family, so I am indebted to them.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>ÖZ</td>
<td>v</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>vii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>xiv</td>
</tr>
</tbody>
</table>

# CHAPTERS

1. INTRODUCTION                  | 1   |
   1.1 Motivation                | 1   |
   1.2 Scope and Goal            | 2   |
   1.3 Contributions of the Study| 2   |
   1.4 Structure of the Thesis  | 3   |

2. BACKGROUND                   | 5   |
   2.1 Discovery of DNA         | 5   |
   2.2 Sequencing               | 5   |
2.2.1 PCR

2.2.2 Sanger sequencing

2.2.3 Next Generation Sequencing (NGS)
  2.2.3.1 Second Generation Sequencing
  2.2.3.2 Third Generation Sequencing

2.2.4 Gene sequence file types

2.3 DNA Barcoding

2.4 Machine Learning
  2.4.1 Underfitting
  2.4.2 Overfitting

2.5 Neural Networks
  2.5.1 Encoding
  2.5.2 Gradient Descent
    2.5.2.1 Vanishing Gradients Problem
    2.5.2.2 Exploding Gradients Problem
  2.5.3 Hyperparameters in Deep Learning
    2.5.3.1 Model (Structure) Related Parameters
    2.5.3.2 Training (Optimization) Related Parameters
  2.5.4 Specialized Neural Networks

2.6 Related Work

3 MATERIAL AND METHOD
  3.1 Data Preparation
  3.2 Model structure
5.4 Specifications of the Output Layer .................................................. 31
5.5 Multi-species Classification .............................................................. 32
5.6 Comparison to similar work .............................................................. 32
5.7 Conclusion ....................................................................................... 33
5.8 Future Work ................................................................................... 34

REFERENCES ...................................................................................... 35

APPENDICES

A CONTENTS OF FILE PREP.PY .......................................................... 39
B CONTENTS OF FILE TRAIN_TEST.PY ............................................... 41
C LIST OF INSTALLED PACKAGES AND VERSIONS .......................... 45
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequencing Technologies and Specifications</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Result Comparison on Different Test and Validation data setups</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Accuracy Test for Different Activation Functions</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Accuracy Test for Different Activation Functions on Output Layer</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>Accuracy Test for Different Sampling Methods regarding Species Representation</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>Decreased Loss Reduction in Vanishing Gradient Problem</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Packages in environment ( tf_{gpu} )</td>
<td>45</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1</td>
<td>Sanger Sequencing Pipeline</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Example FASTA file</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Example FASTQ file</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Neural Network Activation</td>
<td>10</td>
</tr>
<tr>
<td>Figure 5</td>
<td>One-hot Encoding for Hair Color Data</td>
<td>10</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Dummy Encoding for Hair Color Data</td>
<td>11</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Label Encoding for Weight Data</td>
<td>11</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Label Encoding for non-ordinal Hair Color Data</td>
<td>11</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Encoding of the Data</td>
<td>18</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Overview of the Neural Network</td>
<td>18</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Testing Number of Hidden Layers</td>
<td>21</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Test of Layer Sizes</td>
<td>22</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Test of Learning Rates on Read Length 50</td>
<td>23</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Test of Learning Rates on Read Length 500</td>
<td>23</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Test of Standard Deviations for Initializer</td>
<td>25</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Test of Standard Deviations for Initializer on read length 500</td>
<td>25</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Test of Means for Initializer</td>
<td>26</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Visual representation of different sampling cases in case of the inclusion of Drosophila melanogaster as a third species</td>
<td>30</td>
</tr>
</tbody>
</table>
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>WGS</td>
<td>Whole Genome Sequencing</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>RNA</td>
<td>Ribonucleic Acid</td>
</tr>
<tr>
<td>FNN</td>
<td>Feedforward Neural Networks</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Networks</td>
</tr>
<tr>
<td>NN</td>
<td>Neural Networks</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Networks</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Networks,</td>
</tr>
<tr>
<td>PCR</td>
<td>Polymerase Chain Reaction</td>
</tr>
<tr>
<td>NGS</td>
<td>Next Generation Sequencing</td>
</tr>
<tr>
<td>LSTM</td>
<td>Long Short Term Memory</td>
</tr>
<tr>
<td>GRU</td>
<td>Gated Recurrent Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>PCR</td>
<td>Polymerase Chain Reaction</td>
</tr>
<tr>
<td>ddNTP</td>
<td>Dideoxyribonucleotide</td>
</tr>
<tr>
<td>NaN</td>
<td>Not a Number</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Identifying species has been around since there were cave paintings. Primitive classification criteria included; size, colour, number of limbs, mobility etc. It also was limited to the species the people encountered, which was not much. The first great generalizer in classification was Aristotle, who virtually invented the science of logic, of which classification was a part for 2,000 years. The Aristotelian method dominated classification until the 19th century. Carolus Linnaeus, who is usually regarded as the founder of modern taxonomy and whose books are considered the beginning of modern botanical and zoological nomenclature, drew up rules for assigning names to plants and animals and was the first to use binomial nomenclature consistently. [1] There was something that differentiated all the species but it was yet unknown.

Discovery of DNA was a big step in uncovering this unknown factor. This was followed by the discovery of the central dogma. Genes were identified and linked to proteins. These proteins were observed inside and outside of the cells and were found to be responsible for almost every single function any living organism has.

In later years, the world continued to embrace digital technologies and the field of bioinformatics emerged. Computers offered analysis humankind could not perform by hand, and provided visualization that made concepts easier to understand. As the sequencing technologies evolved and costs reduced, getting a species’ whole genome became a regular task. This led to the collection of a large amount of genomic data.

The advancement in digital technologies in other fields entailed the collection of large amounts of data in those fields as well. This led to the introduction of the term big data. The sheer size of big data led to the development and refinement of machine learning methods. Genomic data, despite not being macro-scale, has been classified as big data due to its immense size. Advancement in the classification and compression of such data is critical to allow scientific progress in bioinformatics.

1.1 Motivation

The task of species classification has been tackled in the past in a few ways. The most basic solution was the alignment of sequences to compare differences and to determine a species. In order to be deterministic, this required the use of a large sequence.
Then Hebert et al.\cite{2} proposed a system where one would select a specific position in the sequence where the nucleotides differed interspecies but were mostly unchanged intraspecies. Then this gene could be used as a DNA barcode. Some genes that fit this criterion were found for animals\cite{3}(COI, Cytob, 12S, 16S). Most of these were mitochondrial genes and were selected because they lacked intronic regions, were haploidically inherited, and had limited recombination. For some kingdoms other than the animal kingdom, a combination of genes was selected to successfully apply this technique. For some others, no such combination was found. The weakness of using DNA barcoding was the requirement of a reliable reference library, which was extremely difficult to create in a comprehensive manner. It also lacked a standard protocol to use.

The use of deep learning allowed the classification task itself to evolve from basic alignment of the sequences to computing distances and training neural network models. These models were trained most commonly on the genes that were formerly selected to be used in DNA barcoding, since such genes were already shown to be present in multiple species and to be mostly unique interspecies. However, this method was still held back by the limitations of DNA barcodes, because they were performing the classification by using only certain parts of the DNA.

DNA barcoding requires either annotated data or large sequences to work, so that an alignment can be performed to determine the barcode area in the sample. But second generation sequencing machines can output sequences ranging from 75 to 400 base long sequences. Therefore we need a network that can classify reads of these length.

1.2 Scope and Goal

In this study we are taking the whole reference genomes for two species: Saccharomyces Cerevisiae (Baker’s Yeast) \cite{6} and Escherichia Coli (E. Coli) \cite{7} and training a deep learning model to identify short reads(50-500 bases) taken from the reference genomes using an artificial neural network. The goal is to successfully identify the species of any short read we have sampled out of the reference genomes themselves.

1.3 Contributions of the Study

This study provides a neural network model that can handle the task of binary classification using short(50-500 bp) reads taken from the reference genomes of two species to successfully(\sim 83% accuracy for 50 bp, \sim 91% accuracy for 500 bp) classify similarly generated reads.

This network can be extended to classify more than two species. In which case, it can prove to be a strong tool to classify certain reads where other methods in the literature fall short. Classifying regions where comprehensive DNA barcode information is lacking has not been explored extensively. Research on short reads of non-barcode regions is even less common. For some domains viable DNA barcodes have not yet been discovered.
1.4 Structure of the Thesis

Chapter 2 contains the background information for the subjects included in the thesis. Chapter 3 considers the specifications of the neural network model and the details of the training process. Chapter 4 presents the test results. Chapter 5 discusses the results, compares this study to other similar studies and offers the conclusions of the thesis and possible future directions. In Appendices A and B are the python code files used to prep the data and build, train and test the neural network, respectively. In appendix C is the table with software versions installed in the conda environment.
CHAPTER 2

BACKGROUND

2.1 Discovery of DNA

The origins of the discovery of hereditary similarities date back as far as Ancient Greece, where Hip-
pocrates built a foundation for what is now known as Charles Darwin’s Pangenes(1868) [8]. Mendel’s
work(1866) [9] solidified the existence of a genetic structure housed within living creatures and is
passed down using a dual encoded method to offsprings. Around the same time, Meisner discovered
an irregular phosphorus-rich compound, unlike any protein, calling it nuclein. This nuclein would later
go on to be called DNA(Deoxyribonucleic acid). With the help of X-ray crystallography and former
various theories, Watson and Crick theorized a shape for what this genetic material could be, giving
birth to the double-helix structure DNA is known to be today [10].

2.2 Sequencing

Sequencing is the process of experimentally determining the sequence of bases that construct an or-
ganism’s genetic materials. There are different methods used to sequence the genetic materials, an
extensive list of used machines can be found in Table 1.

2.2.1 PCR

It is a commonly used molecular biology technique that aims to make multiple copies of a region of
DNA. A double-stranded DNA molecule is broken apart and is built back together using the primers
that are provided. Therefore effectively replicating the DNA molecule.

2.2.2 Sanger sequencing

Also known as the chain termination method, the dideoxynucleotide method, or the sequencing by
synthesis method. First discovered by Frederick Sanger[11] and his colleagues in 1977. It played an
important role in the Human Genome Project.

As illustrated in Figure 1[12], Sanger Sequencing is composed of three steps.
<table>
<thead>
<tr>
<th>Generation</th>
<th>Platform</th>
<th>Instrument</th>
<th>Reads per run</th>
<th>Avg Read Length (pb)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Generation</td>
<td>ABI Sanger</td>
<td>3730xl</td>
<td>96</td>
<td>400–900</td>
<td>2002</td>
</tr>
<tr>
<td>454</td>
<td>GS20</td>
<td>200</td>
<td>100</td>
<td>2005</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>GS FLX</td>
<td>400</td>
<td>250</td>
<td>2007</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>GS FLX Titanium</td>
<td>1M</td>
<td>450</td>
<td>2009</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>GS FLX</td>
<td>1M</td>
<td>700</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>Titanium+</td>
<td>1M</td>
<td>700</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>GS Junior</td>
<td>100</td>
<td>400</td>
<td>2010</td>
<td></td>
</tr>
<tr>
<td>454</td>
<td>GS Junior+</td>
<td>100</td>
<td>700</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>Second Generation</td>
<td>Illumina MiniSeq</td>
<td>25M (max)</td>
<td>150</td>
<td>2013</td>
<td></td>
</tr>
<tr>
<td>Illumina</td>
<td>MiSeq</td>
<td>25M (max)</td>
<td>300</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>Illumina</td>
<td>NextSeq</td>
<td>400M (max)</td>
<td>150</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>Illumina</td>
<td>HiSeq</td>
<td>5B (max)</td>
<td>150</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>Illumina</td>
<td>HiSeq X</td>
<td>6B (max)</td>
<td>150</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>SOLiD</td>
<td>5500 W</td>
<td>3B</td>
<td>75</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>SOLiD</td>
<td>5500xl W</td>
<td>6B</td>
<td>75</td>
<td>2013</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>PGM 314 chip v2</td>
<td>400,000-550,000</td>
<td>400</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>PGM 316 chip v2</td>
<td>2M-3M</td>
<td>200</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>PGM 318 chip v2</td>
<td>4M-5.5M</td>
<td>400</td>
<td>2013</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>Ion Proton</td>
<td>60M-80M</td>
<td>200</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>Ion S5/S5XL 520</td>
<td>3M-5M</td>
<td>400</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>Ion S5/S5XL 530</td>
<td>15M-20M</td>
<td>400</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Ion Torrent</td>
<td>Ion S5/S5XL 540</td>
<td>60M-80M</td>
<td>400</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Third Generation</td>
<td>PacBio RS C1</td>
<td>432</td>
<td>1300</td>
<td>2011</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>RS C2</td>
<td>432</td>
<td>2500</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>RS C2 XL</td>
<td>432</td>
<td>4300</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>RS II C2 XL</td>
<td>564</td>
<td>4600</td>
<td>2013</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>RS II P5 C3</td>
<td>528</td>
<td>8500</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>RS II P6 C4</td>
<td>660</td>
<td>13500</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>PacBio</td>
<td>Sequel</td>
<td>350</td>
<td>10000</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Oxford Nanopore</td>
<td>MinION Mk</td>
<td>100</td>
<td>9545</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Oxford Nanopore</td>
<td>PromethION</td>
<td>NA</td>
<td>9846</td>
<td>2016</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Sequencing Technologies and Specifications
• 1) PCR with fluorescent, chain-terminating ddNTPs

Chain-terminating PCR is applied. Similar to standard PCR the goal is to replicate the sequence, but with one major difference: the addition of modified nucleotides (dNTPs) called dideoxyribonucleotides (ddNTPs). In the extension step of standard PCR, DNA polymerase adds dNTPs to a growing DNA strand by catalyzing the formation of a phosphodiester bond between the free 3’-OH group of the last nucleotide and the 5’-phosphate of the next.

• 2) Size separation by capillary gel electrophoresis

The oligonucleotides are put in a gel matrix and applied an electrical current. Because DNA is negatively charged, the oligonucleotides are pulled, at different speeds due to their size. This helps order them by size.

• 3) Laser excitation & detection by sequencing machine

Since DNA polymerase starts its work from the provided primer, by reading the gel bands from smallest to largest, the 5’ to 3’ sequence of the original DNA strand can be determined.

This method was revolutionary at its time but proved too slow and too expensive as alternative approaches emerged.

2.2.3 Next Generation Sequencing (NGS)

2.2.3.1 Second Generation Sequencing

Second Generation Sequencing saw the introduction of massively parallel sequencing with Illumina. Soon followed by Ion Torrent. They excelled at sequencing many small parts quickly and at a low cost. For this reason, they excelled at resequencing projects and SNP calling. Their shortcoming was their short reads of ∼ 50 – 200bp.
2.2.3.2 Third Generation Sequencing

Third generation sequencing technologies allow for much longer read lengths.

2.2.4 Gene sequence file types

Gene data has different formats it can be stored in depending on the specific information it contains. It can contain sequence data, annotation data, quantitative data and read alignments.

- **FASTA**[13]
  
  Can be used to represent either nucleotide or amino acid sequences. Can have the following extensions: .fasta, .fna, .ffn, .faa, .frn, .fa.

> BK006935.2 TPA_inf: Saccharomyces cerevisiae S288C chromosome I, complete sequence
ccacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacacac
2.4 Machine Learning

With the increase in processing power of computers, it became possible to analyze whole genomes. Pattern recognition, anomaly detection, clustering, and classification were tasks that could be automated. Which led to even more information to be extracted from genetic materials.

The goal of most machine learning models is to construct a model and fit that model to the task/input at hand in a way that can produce the desired outcome. This fitting process needs to be adjusted according to the results it produces with both its original training set and similar yet non-identical test set. According to its scores on these tests, its fit is determined.

2.4.1 Underfitting

If the machine learning model can not successfully learn, it can be seen on its scores. If the model underperforms in both the training and the test set, it is said to be underfit. Underfitting can be due to an insufficient learning time, a simple model structure or over regularization. It can be overcome by changing these.

2.4.2 Overfitting

If the machine learning model performs well specifically on the set it was trained on, and underperforms in the test set; it is said to be overfit, meaning that it captured the specifications of the training set too well when it was desired to be more accommodating of a wider range of data. The training set could be enriched with more diverse data points.

2.5 Neural Networks

As the technology advanced, humankind analyzed what makes intelligence, or even consciousness, and tried to mimic it. The trials of simulating the biological neural networks of the brain tissue led researchers to Artificial Neural Networks. The core structure of Neural networks is as follows:

- There is an input layer and an output layer. There may be varying numbers of "hidden layers" in between.
- Each layer contains a certain number of nodes. These nodes are connected to other nodes in adjacent layers. This enables the flow of information.
- These connections each have weights and associated activation functions. This manipulates the flow of information to, hopefully, extract, store and infer a certain piece of information that belongs to the input. (This piece of information can be positive/negative for a sentence, containing a cat for an image, belonging to a specific species for DNA in our case)
- These weights are the main holders of information. The other specifications of the model are called "hyperparameters" and are configured according to the specific model structure and input data type used to extract the intended information out of the input data.
Since numbers and mathematical operations are used throughout the model, the values of the output layer can be represented in terms of the values of the input layer, the weights of the connections between nodes, the activation function and a correcting value called bias. By feeding the desired outcome and the input to the model, one can tell the model to resemble that outcome.

In this case, a function to calculate resemblance/difference/distance is needed. This is where the loss function comes in. Using the loss function, one can determine how alike the model is to the ideal model. The weights can then be adjusted in a way so that the model is more similar to this ideal model. How much change each learning step makes is called the learning rate.

\[ a^{(1)} = \sigma \left( W^{(0)}a^{(0)} + b^{(0)} \right) \]

Figure 4: Neural Network Activation

2.5.1 Encoding

Encoding is used to represent categorical(specifically nominal in our case) data so that it can be used in a mathematical fashion.

- **One-hot Encoding**

  Useful for nominal data where one cannot quantify the difference between the classes. Can also be used for ordinal data with ease, however, capturing the actual distances between values will be out of the question. Increases the dimensions of the data significantly. Easy to quantify differences and calculate distances due to its high dimensional nature. Every class increases the dimensions, which can lead to issues, so not very useful when the classes are high in number. A dummy case can be added where all the values are 0 which would reduce the total number of dimensions by 1.

- **Label/Ordinal Encoding**

  Useful for ordinal data where you can order the classes. Even though the difference/distance between the points will not be accurate when calculated, they will be comparable to each other and will be statistically indicative.

Figure 5: One-hot Encoding for Hair Color Data
Although it is best suited for ordinal data, label encoding can also be used on non-ordinal data. In such a case it is prudent to be aware that the distance between Blonde-Brown and Brown-Grey will be identical and will be the half of Blonde-Grey. This is nonsensical in terms of the data used, so some metrics should not be given attention.

- **Target-Mean Encoding**

  Compared to other encoding types, Target Encoding encodes the data with values that are indicative of the target. Most commonly the mean of an indicative column is used. Can be used in a way where the value is calculated without using that specific value, called the Leave one out method.

- **Frequency/Count Encoding**

  Can be useful in displaying the distribution of a set. Every value is encoded with the count of instances it has in the dataset, or the overall frequency(count/total count).

### 2.5.2 Gradient Descent

A gradient is the partial derivatives of a function at a given point. Gradient descent is an optimization algorithm that tries to minimize the gradient, achieving a local minimum. There are some issues that can be encountered regarding this algorithm:

#### 2.5.2.1 Vanishing Gradients Problem

During backpropagation, gradients are used to update the weights. In this step, if the gradients keep getting smaller where they are taken as zero, the weights will not be updated since the algorithm believes it has reached a local minimum point, when in fact it has not. As a result, the gradient descent algorithm never converges to the optimal solution. Can be spotted by slow or non-existent learning process\[14\]
2.5.2.2 Exploding Gradients Problem

On the other hand, if the gradients become so large they can make the gradient descent diverge instead. This can occur when the denominator approaches zero, overflowing the gradient, making it NaN at which point it can no longer be worked on. Can be spotted by large changes at every step or even NaN valued losses.

2.5.3 Hyperparameters in Deep Learning

A big part of the machine learning process is fine-tuning everything. Split into two model structure parameters and training (learning) parameters

2.5.3.1 Model (Structure) Related Parameters

- **Number of hidden layers/neurons**
  A neural network model with more than three layers (one input layer, one output layer, one or more hidden layers) is considered a deep learning method. The number of hidden layers may depend on the number of features desired to be captured from the input data. Also increasing the number of hidden layers and neurons tend to increase the accuracy, but can eventually cause overfitting, just as too few layers can cause underfitting.\[15\]

- **Dropout**
  The models are trained on specific data, but often times are expected to be effective on similar yet not identical data. For this reason, it is essential not to overfit. Dropout is a method to overcome overfitting. Some nodes in the model are temporarily removed from the model so that some randomness (noise) can be added into the training process. The goal is to break up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust.\[16\]

- **Momentum**
  Momentum allows the learning process to easily get to where it needs to. It achieves this by adding the previous update vector into the next batch. This allows for easier convergence.

- **Weight initialization**
  There are a few options that can be used to initialize the weights between the nodes. Assigning all zeros or ones is an option, or to add some noise, like before, random values can be assigned. But the general vicinity of this randomness can be controlled, using Normal or Uniform distributions to select the random values. The specifications of the distributions (mean, standard deviation, min, max) are then become the hyperparameters to tune the models

- **Activation Function**
  As can be seen on Figure 4 the activation function ($\sigma$) is used in the calculation of the nodes throughout the learning process. They are used to introduce non-linearity to the model. Different activation functions used:
- Tanh
  \( \tanh(x) \). Provides values between (-1,1).

- Sigmoid
  \[ \frac{1}{1+e^{-x}} \]

- Rectified Linear Units (ReLU)
  Offers a non-negative value by taking the max(0, x). Produces values between [0, inf)

**Loss function**

The loss function determines the fit of the model. The goal of training the model is to minimize the difference between the desired output and the actual output of the model. The loss function helps us achieve that. During training the loss function eventually leads into a local minimum. The local minimum is desired to be a global minimum. Commonly used loss functions are as follows:

- Mean Squared Error
  MSE is calculated by taking a mean of the squares of the distances between points in the data.

- Cross-entropy
  Useful for calculating the distance(difference) between categorical data points. Cross-entropy loss is also called logarithmic loss, log loss, or logistic loss. Each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0. [17] Cross-entropy is expressed by the equation;

  \[
  l(p, q) = - \sum p(x) \log q(x) \quad (1)
  \]

  Where \( x \) represents the predicted results by ML algorithm, \( p(x) \) is the probability distribution of the “true” label from training samples, and \( q(x) \) depicts the estimation of the ML algorithm.

  * Categorical Cross-entropy is used for classification tasks where multiple classes are present.
  * Binary Cross-entropy is used in binary classification tasks.

2.5.3.2 Training(Optimization) Related Parameters

- **Learning rate**
  Learning rate indicates how much correction will be made in response to the error calculated by the loss function after each epoch.

- **Number of epochs**
The number of epochs determine how many times the training process will be repeated on the training data. To achieve convergence, the training process may need to be repeated a number of times. The rule of thumb to determine the correct number of epochs is to stop when the validation error starts to increase, which can be an indicator of overfitting.

- **Batch size**
  
The batch size determines how many data runs the model will go through before it updates its weights.

### 2.5.4 Specialized Neural Networks

In addition to the core design of neural networks explained above, there are more specialized versions that are more adept at extracting/storing certain patterns in data.

- **Convolutional Neural Networks (CNN)**
  
  As neural networks tried to mimic the process of learning, Fukushima (1980)[18] tried the same for the human visual pattern recognition capability. The previous models lacked the ability to handle inputs of varying shapes. But by moving the learning process through the input this obstacle was overcome. Like the animal visual cortex of the human body, the neurons correspond to multiple points of the input that are adjacent to them and store them as a single representative value. By doing this with multiple neurons, the whole data can be covered.

  A further simplification process called pooling usually follows. In pooling multiple points are reduced to one neuron mainly by `max()` or `avg()` functions. By repeating these steps multiple times it is possible for different layers to represent different properties in the input.

  CNNs excel at pattern recognition in a variety of data types, specifically image and video data.

  Most common example encountered when CNNs are concerned is the MNIST database of handwritten digits[19]. In this example, the power of CNNs is obvious. By having multiple convolutional layers, it is possible to capture lines->connected lines->numbers in order.

- **Recurrent Neural Networks (RNN)**
  
  RNNs use sequential or time series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning. They are distinguished by their “memory” as they take information from prior inputs to influence the current input and output. The output of recurrent neural networks depend on the prior elements within the sequence. There are bidirectional RNNs that also take into account the outputs that would follow this output in order.

  RNNs share parameters across each layer of the network which need to be tuned similarly to feedforward networks. The weights on the RNNs are adjusted through the processes of backpropagation and gradient descent to facilitate reinforcement learning.

  RNNs utilize a process called backpropagation through time (BPTT) where the errors it has is summed throughout the training process. Due to this method, RNNs can easily encounter two problems, known as vanishing gradients and exploding gradients. These issues are defined by the size of the gradient, which is the slope of the loss function along the error curve. Where the weights are either too small or too large. In which case the weights become insignificant or
not a number (NaN) respectively. One solution to these issues is to reduce the number of hidden layers within the neural network, eliminating some of the complexity in the RNN model.\cite{20}

- Long-Short Term Memory (LSTM) \cite{21}
  Developed to combat the vanishing/exploding gradient problems, LSTMs contain a forget gate, which can help the model learn better. \cite{22}

- Gated Recurrent Unit (GRU) \cite{23}
  GRUs are simpler versions of LSTMs, they have fewer gates (just a reset and update gates instead of the forget, input, and output gate of LSTM) therefore fewer parameters but they are less powerful and adaptable.

- **Autoencoders** \cite{24}
  Autoencoders are similar to Artificial Neural Networks in that they essentially have the same architecture. The difference in Autoencoder architecture is that their network always resembles an hourglass like shape, with layers getting smaller closer to the middle.

  The first half of the network is called the encoding part and the second part the decoding part. Backpropagation is used to train the network. They are symmetrical and during the training process the error is calculated as the difference of the output to the input trying to achieve true symmetry. This means the goal is to encode (compress) the input to fit the smallest middle hidden layer(s) to then be able to recreate the input as the output.

  - Variational Autoencoders \cite{25} VAEs have the same architecture as AEs but they use the approximated probability distribution to train.

  \section*{2.6 Related Work}

Abd–Alhalem et al.\cite{26} compares DNA sequence classification algorithms that use deep learning. They split the algorithms in three: alignment based methods, alignment free methods and ML-DSP (Machine Learning - Digital Signal Processing). They point out the rise in popularity of deep learning in DNA sequence classification due to their remarkable performance and reduction of challenges. They emphasize the need for a high amount of data for deep learning to perform successfully.

Rizzo et al.\cite{27} use 16S gene sequences to taxonomically classify bacteria species down to the genus level. They randomly select 1000 sequences from each of the three most common bacteria phyla, Actinobacteria, Firmicutes, Proteobacteria, collecting in total 3000 sequences. All the sequences have length greater than 1200 bp, They use short reads of length 500 bases. They use a Convolutional Neural Network on spectral representations of the genomes. They have very high accuracy on higher (Phylum, class) taxonomic levels, but as they approach the genus level, they require the use of full length sequences to achieve $> 50\%$ accuracy. But when using full length sequences, they achieve $> 85\%$.

Busia et al.\cite{28} also use 16S gene sequence but for a much wider range of species. They also use a CNN. They have 19,851 sequences with 13,838 from 2,768 genera. They tested their model on read lengths $L = 25, 50, 100, 150, 200$. They used 16S mock community sequencing data and they also test on synthetic data they simulate base-flipping noise on. Their results suggest that their approach of injecting random base-flipping noise at train time allows the DNNs to learn a more robust noise
model than the one implicitly incorporated by aligners like BLAST and BWA by allowing gaps and mismatches to be tolerated. Overall, this study acts as a first step towards their long-term goal of developing a general-purpose deep learning model that can successfully perform any task framed as the assignment of labels to short biological sequences.

Weitschek et al. [29] introduce BLOG (Barcoding with LOGic) 2.0 it uses a supervised machine learning approach that extracts species-specific positions of the DNA Barcode sequences from the training set and formulates rules on these positions. Since it is learning its rules on specific positions of the data, it requires a complete reference library of polymorphisms for each species in the training set to avoid false negatives. They use DNA barcode sequence files in FASTA format that are either of the same region or have been pre-aligned. They achieve 92% accuracy on fungi and 100% accuracy on algae.

Yang et al. [30] use DNA barcode sequences to classify species. They use convolutional methods, treating one-hot encoding as its own dimension and convoluting and pooling in 2D. Afterwards taking the flattened pooling results through hidden layers. Using ReLu as its general activation function and softmax in the last layer to compute the probabilities of the species.

Cui et al. [31] propose a genomic sequence compression algorithm based on a deep learning model and an arithmetic encoder. Their deep learning model is structured as a convolutional layer followed by an attention-based bi-directional long short-term memory network, which predicts the probabilities of the next base in a sequence. The arithmetic encoder employs the probabilities to compress the sequence.
CHAPTER 3

MATERIAL AND METHOD

The neural network model was implemented using the Keras toolkit, which is based on TensorFlow libraries in Python programming language. The data is prepared and dumped into files using pickle. This preparation step will especially be useful should there be new genes added to the dataset. Because, since it won’t be possible to fit all the data in the memory at once, batches will need to be implemented due to the increase in the space required. As the species selected had exceptionally small DNA lengths, any addition to the dataset (which would likely involve species with longer DNA) will significantly increase the size.

A virtual environment was used to perform the training and testing using Anaconda. We used its package and environment manager for the command line interfacing tool Conda(version 4.12.0). The list of installed relevant packages and their versions can be found on Table C.

3.1 Data Preparation

Two reference genomes were used to train and test the model: Saccharomyces Cerevisiae (Baker’s Yeast) and Escherichia Coli (E. coli). These species were selected mainly due to their small sized and more importantly well-studied DNA sequences. They are also taxonomically very different which makes them ideal candidates for ascertaining the viability of using the whole genome for species classification. Saccharomyces cerevisiae is a eukaryote whereas eschericia coli is a prokaryote. We wanted to span a wider range of species. The genomes were extracted out of their FASTA files and preprocessed to contain nothing but the genome sequences.

E. coli reference genome consists of 4,699,673 nucleotides and is 4.48 MB in size.

Baker’s Yeast reference genome consists of 12,309,078 nucleotides and is 11.7 MB in size.

Uniform distribution is used to randomly select short reads of length 50 to 500 out of the DNA data. Equal number of samples are collected out of both samples. This decision is discussed further in 5.2.

The files were reformatted to contain only lowercase letters for a unified representation of each nucleotide (thus disregarding the information for repeating sequences indicated by the uppercase letters). Then the data is encoded into a one-hot encoding format. With the special case of the character "N" which can be used to represent the existence of a non-specific nucleic acid, 5 categories were used for the encoding process. The transformation matrix is shown on Figure 9.
3.2 Model structure

The model consists of six layers as can be seen on Figure 10. One input layer, one flatten layer, three hidden dense layers, and one output layer.

The data is taken through a Flatten() layer to reformat the data back into a 1D format. The model consists of three fully connected hidden layers. The hidden layers are separated by activation layers.
3.3 Hyperparameters

The number of layers determines the information that can be extracted from the data. We initially selected three hidden layers for our model. When we tested whether lower number of layers was feasible, we saw that three layers gives us the best results (Figure 11).

We selected our output layer to have two nodes, one for each species. After every test case we check the output layer and select the highest value to be the result of our classification. We can also compare the outputs of our network to determine how confident the classification has been.

We use initializers to assign certain values to the weights in the network to jump-start the learning. We used Keras’ initializer that samples Random Normal distribution. Its default specifications are \text{mean}=0.0, \text{stddev}=0.05, further tests on these values were performed. The values that gave the best results were 0.09 for standard deviation. The mean did not seem to affect the model in a sensible way. For this reason the mean was kept as 0.0 since it gave as good a result as any value (4.1.6).

The activation functions determine the flow of information in the network. The activation functions we considered for our network are as follows: Tanh, Sigmoid, ReLU, Exponential. We opted to use Tanh activation function as trying different activation functions proved difficult with our setup (4.1.4). We tried different activation functions for the last layer, the one that improved the accuracy best was ReLU (4.1.5).

Loss functions help the network learn by providing a metric with which can be optimized to get the desired outcome. Because we have categorical data, the loss functions we considered were: Categorical Cross-entropy and Binary Cross-entropy. We used categorical cross-entropy initially since it allowed us to later add new classes. We then found out that using categorical cross-entropy for binary classification resulted in us getting inconsistent results (further discussed in 5.1.1).

The learning rate determines how much each batch is going to affect the model. It can be adjusted to overcome certain convergence issues. We used Keras’ Adam optimizer which defaulted to 0.001 learning rate. As we tested multiple learning rates, 0.0006 was seen to provide better results (4.1.3).

3.4 Training

Initially, we set a seed to achieve reproducibility of the results. We used \texttt{tf.keras.utils.set_random_seed (seed)} since it sets seeds for Python, Numpy and TensorFlow's randomized operations. Despite setting these seeds, some variance between runs persisted. Further investigation revealed that using the GPU introduced some randomness due to order of operations. To circumvent this issue \texttt{tf.config.experimental .enable_op_determinism()} was used. This option makes the training process deterministic, eliminating some parallel qualities therefore reducing the overall performance of the training process.

The time required to train the model ranges from 15 seconds to 250 seconds with five epochs. The main factor affecting the training times is \textit{Read Length}. We take our reads in a way that the total length of the reads will be double the size of the original DNA. Therefore as the read length increases number of reads decrease. With less reads to train, the training process naturally takes shorter. We were unable
to find a direct correlation of anything else with the training time. This could be due to the noise in the training time data since the other processes running in the device can alter the training times.

We tried three devices throughout the project. Even though the devices had similar setups, the training times were changed tenfold. Rather than investigate the cause of this issue, we continued using the fastest device.

3.5 Testing

The testing process is aimed towards adjusting the hyperparameters in the network. We used Accuracy Score [Equation 2] and F1 Score [Equation 3] to measure test success. F1 score is effective in indicating a class imbalance in some cases where accuracy seems to have increased.

We take the Baker’s Yeast as the positive outcome in our tests. This is due to it being the first species in the species list.

\[
\text{Accuracy} = \frac{\text{TruePositives} + \text{TrueNegatives}}{\text{TruePositives} + \text{FalsePositives} + \text{TrueNegatives} + \text{FalseNegatives}} \quad (2)
\]

\[
F1 = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 * \text{TruePositives}}{2 * \text{TruePositives} + \text{FalsePositives} + \text{FalseNegatives}} \quad (3)
\]

We use `sklearn.model_selection.train_test_split(test_size=0.20)` to reserve 20% of our initial data to use for validation and testing. The validation set is fed to the network during its training. After the training is done, we run this test data through Keras’ `predict()` function to get its predictions. By comparing these predictions to the actual classes we can judge the performance of our model.

After all of our tests was performed, it was pointed out that our validation test being the same as our test set could lead to overfitting. So we compared the results of an 80/20/20 split and an 80/10/10 split. The results can be seen on Table 2. It is important to point out that in these cases when splitting the data into 80/20/20 the validation set and the test set becomes identical but when splitting the data into 80/10/10 the validation set and the test set are separate. The results of this adjustment didn’t prompt us to perform all of our previous tests again since it had a minimal effect on the network results where accuracy and F1 scores are concerned.

<table>
<thead>
<tr>
<th>Train Set</th>
<th>Test Set</th>
<th>Validation Set</th>
<th>Accuracy</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>20%</td>
<td>20%</td>
<td>0.894</td>
<td>0.886</td>
</tr>
<tr>
<td>80%</td>
<td>10%</td>
<td>10%</td>
<td>0.896</td>
<td>0.888</td>
</tr>
</tbody>
</table>

Table 2: Result Comparison on Different Test and Validation data setups
4.1 Experiments

4.1.1 Testing Number of Hidden Layers

As an initial decision, the number of hidden layers was set to be three. In this experiment we tested different numbers of hidden layers in a similar model structure. In Figure 11 we can see their highest performing specifications’ accuracy values.

Figure 11: Testing Number of Hidden Layers
4.1.2 Testing Read Length and Layer Sizes

As can be seen on Figure 12, the increase in the length of the reads of DNA we used correlates with an increase in accuracy, which was to be expected. However, since the goal was to find solutions for short reads of data, reads of 50 were kept in all other tests.

Layer sizes of 200/100/20 was found to be the best performing for read length 50. 1000/100/20 for 500 read length.

Figure 12: Test of Layer Sizes

4.1.3 Testing Learning Rates

Figure 13 clearly shows an ideal range of learning rate values to use([0.005, 0.01]). As the values differ from this range the performance of the model decreases.

On a network of input size(read length) 500, the optimal learning rate seemed to be much lower than the initial values as Figure 14 shows. We selected 0.00003 to be the optimal value, since it was surrounded by more consistent results.
Figure 13: Test of Learning Rates on Read Length 50

Figure 14: Test of Learning Rates on Read Length 500
4.1.4 Testing Different Activation Functions

Table 3 has the results of the different activation function test. The only viable option here is tanh. The other cases are due to vanishing and exploding gradients. This is further discussed in chapter 5.

<table>
<thead>
<tr>
<th>Function</th>
<th>Accuracy</th>
<th>F1</th>
<th>Epoch 1</th>
<th>Epoch 2</th>
<th>Epoch 3</th>
<th>Epoch 4</th>
<th>Epoch 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>0.8361</td>
<td>0.8266</td>
<td>0.5786</td>
<td>0.5485</td>
<td>0.5390</td>
<td>0.5353</td>
<td>0.5330</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Val Loss</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5568</td>
<td>0.5442</td>
<td>0.5426</td>
<td>0.5359</td>
<td>0.5362</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>0.4989</td>
<td>0.6657</td>
<td>0.6931</td>
<td>0.6931</td>
<td>0.6931</td>
<td>0.6931</td>
<td>0.6931</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Val Loss</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.6932</td>
<td>0.6932</td>
<td>0.6932</td>
<td>0.6932</td>
<td>0.6932</td>
</tr>
<tr>
<td>ReLU</td>
<td>0.5017</td>
<td>0.6682</td>
<td>0.6268</td>
<td>0.6134</td>
<td>0.6087</td>
<td>0.6064</td>
<td>0.6050</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Val Loss</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.6172</td>
<td>0.6123</td>
<td>0.6089</td>
<td>0.6078</td>
<td>0.6074</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.5002</td>
<td>0.6668</td>
<td>Loss</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Val Loss</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Table 3: Accuracy Test for Different Activation Functions

4.1.5 Testing Different Activation Functions for the Output Layer

<table>
<thead>
<tr>
<th>Activation Function</th>
<th>Accuracy</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>tanh</td>
<td>0.827</td>
<td>0.821</td>
</tr>
<tr>
<td>sigmoid</td>
<td>0.827</td>
<td>0.826</td>
</tr>
<tr>
<td>relu</td>
<td><strong>0.835</strong></td>
<td><strong>0.829</strong></td>
</tr>
<tr>
<td>exponential</td>
<td>0.828</td>
<td>0.826</td>
</tr>
<tr>
<td>softmax</td>
<td>0.829</td>
<td>0.826</td>
</tr>
</tbody>
</table>

Table 4: Accuracy Test for Different Activation Functions on Output Layer

Table 4 shows us that a final layer with ReLu activation performs slightly better.

4.1.6 Testing Initializer Specs for Normal Distribution

4.1.6.1 Standard Deviation

The standard deviation values `keras.initializers.RandomNormal(stddev, mean)` uses to set the initial node weights seems to have a range in which it lets the network train the best [0.07, 0.09]. (Figure 15)

When testing on read length 500 reads we found [0.03, 0.05] to be the optimal range of values as can be seen on (Figure 16)
Figure 15: Test of Standard Deviations for Initializer

Figure 16: Test of Standard Deviations for Initializer on read length 500
4.1.6.2 Mean

The mean values `keras.initializers.RandomNormal(stddev, mean)` uses to set the initial node weights doesn’t seem to have a significant effect on the model’s performance. Figure 17

![Figure 17: Test of Means for Initializer](image)

4.1.7 Testing Different Sample Species Representation Methods

In Table 5 Different representation of species in the samples were investigated. This test was conducted for two different read lengths: 50 and 500. When the samples are taken proportional to the length of the reference genomes of the species, the accuracies seem to increase. But the decrease in F1 score show us that this is in fact not the case. This leads us to think that there is a big imbalance of results between classes/species.

<table>
<thead>
<tr>
<th>Read Length</th>
<th>Number of Samples</th>
<th>Accuracy</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ecoli</td>
<td>Yeast</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>18566</td>
<td>486284</td>
<td>0.869</td>
</tr>
<tr>
<td></td>
<td>486284</td>
<td>486284</td>
<td>0.836</td>
</tr>
<tr>
<td>500</td>
<td>18566</td>
<td>48628</td>
<td>0.919</td>
</tr>
<tr>
<td></td>
<td>48628</td>
<td>48628</td>
<td>0.903</td>
</tr>
</tbody>
</table>

Table 5: Accuracy Test for Different Sampling Methods regarding Species Representation
We have explained our network and presented our results in the previous chapters. Now we will talk about some of the issues we have encountered throughout the development process and some things to consider in further development.

5.1 Issues Encountered

5.1.1 Loss Selection

We were stuck randomly getting inverted accuracy scores. This issue persisted for a long time. The accuracy would, seemingly randomly, switch between \( \sim 80\% \) and \( \sim 20\% \).

This looked similar to a labelling issue we previously had where the labels of the classes were mixed and we were getting all the results as \( \sim 20\% \). But the differences were that the switch between positive and negative scores were at random and not in batches unlike our previous problem. Also the Accuracy scores weren’t exactly complementary. In our previous problem the labels in the test set were mixed so the training process was untainted, which led to accuracy scores that would add exactly up to 1. But that wasn’t the case here.

During random tests we discovered that the issue resulted from the use of \( \texttt{tf.keras.losses.CategoricalCrossentropy} \). We had used Categorical Crossentropy so we could, in the future, add more classes to the network. But as it stands our task is binary classification. Our assumption was that both Binary Crossentropy and Categorical Crossentropy would give us the same results in a binary classification task. Apparently Categorical Crossentropy is not suitable for use with only two classes. A switch to \( \texttt{tf.keras.losses.BinaryCrossentropy} \) resolved our issue.

5.1.2 Gradients in Activation Function Selection

During our tests to find the most suitable activation function for the network, we encountered some nonsensical results. As can be seen on 4.1.4 Tanh function was the only feasible option. We think these nonsensical results are due to vanishing and/or exploding gradients.

As explained in 2.5.2, a vanishing gradient problem can be spotted by a slow or non-existent learning process. For our test on Sigmoid function, we can see from the losses that the network hasn’t learned
at all. In case of ReLU function, as illustrated in Table 6 when compared to the first epoch of Tanh function’s losses, the reduction in loss (which usually indicates learning) is very low. We can deduct from this slow loss reduction and low accuracy that the model doesn’t seem to be learning, ∼ 50% accuracy that the gradients are vanishing. For Exponential function the losses are all NaN which is probably an indicator of integer overflow. This means that the value of the loss has become so high that the value can not be represented by the programming language. This is most probably due to exploding gradients. Gradient clipping can be performed to overcome this issue but since Tanh is a perfectly acceptable option, we did not explore the other options further.

<table>
<thead>
<tr>
<th>Function</th>
<th>Epoch 1</th>
<th>Epoch 2</th>
<th>Epoch 3</th>
<th>Epoch 4</th>
<th>Epoch 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>0.5786</td>
<td>0.5485</td>
<td>0.5390</td>
<td>0.5353</td>
<td>0.5330</td>
</tr>
<tr>
<td>ReLU</td>
<td>0.6268</td>
<td>0.6134</td>
<td>0.6087</td>
<td>0.6064</td>
<td>0.6050</td>
</tr>
</tbody>
</table>

Table 6: Decreased Loss Reduction in Vanishing Gradient Problem

However, we continued to observe outcomes similar to those of the vanishing gradient problem. These happened mostly on edge cases while we were performing wide range of values on our hyperparameters. We tracked this particular issue to our loss function. Loss functions in Keras/TensorFlow have from_logit = False as default. This option makes the loss function assume that all the values it gets will be (-1, 1). But since we are using Tanh activation function for intermediary nodes and ReLU for our output layer, therefore not normalizing our weights, we needed to set from_logit = True for our loss function. This makes the loss function apply the softmax function to normalize the output values before it uses them.

5.1.3 Achieving Determinism

As we mentioned before, we used keras.utils.set_random_seed() to set seeds for python, numpy and tensorflow. But we still weren’t able to get deterministic results. Runs with same specifications were providing similar yet non-identical results. Upon further investigation we found that the randomness was due to the randomness introduced by the order of processes in the GPU. To overcome this we set tf.config.experimental.enable_op_determinism(). With this option we achieved true deterministic results. This determinism came at a cost of training time since the GPU was no longer using multiple cores to optimize the process.

5.1.4 Using Keras’ Own Metrics

Keras has its own metrics that can be used to track the learning process. We encountered various errors when trying to use them. We later found these errors to be caused by two things: one of them is because we set tf.config.experimental.enable_op_determinism(), and the other is due to our use of
logits, since these metrics were incompatible with logits. These issues are fixed in newer versions of Tensorflow but due to some dependencies we have, we were unable to upgrade Tensorflow.

5.1.5 Memory/Time Issues in Training

When training our network, our code took a very long time, and in some specifications the device would run out of memory. We then realized that since we were using numpy.append() to read the data into an array, at every step a new array would be created which increased the time and space it took. Creating an empty array first and then filling it resolved this issue.

With the current two species setup, batching isn’t required. Should there be more species added to the dataset, there will become a need for batching the data, since it may not be possible to load all of the data into memory all at once.

Similarly, initially we would read the data file, sample, train and test all in the same file. We then separated the files. prep.py(A) file reads and samples the data. It then dumps the sampled reads into a pickle file. test.py(B) reads the sampled reads from the pickled file, creates and trains the network and tests and outputs the specifications of the test and its results.

As pointed out before, we tried three different devices throughout the project. One of the devices was 10 times slower without any apparent differences that could cause this. We continued the tests using the fastest device.

5.2 Representation of Species in the Data

We initially selected the number of samples for each species to be proportional to their length. This decision was due to the possible addition of different species to the data. Because in such a case, the size of the samples would increase exponentially. The sampling size differences in a hypothetical inclusion of the species Drosophila Melanogaster can be found on Figure 18. Drosophila Melanogaster reference genome consists of 145,523,498 nucleotides and is 138 MB in size therefore the lengths of the genomes in the figure closely represent a real-life case.

In Case A, the space the samples take up is $\sum_{i=0}^{n} l_i$, where $n$ is the number of species in the data, and $l_i$ is the length of the genome of the $i^{th}$ species in the data. With the introduction of the third species Drosophila, $3n$ total data size would increase to $31n$.

However, in Case B, the space the samples take up is $l_{max} * n$ where $n$ is the number of species in the data, and $l_{max}$ is the length of the longest genome. $4n$ total data size would increase to $84n$.

This is indicated in the prep.py(A) file with a flag `species_length_equalized`. 0 for Case A and 1 for Case B.

In Table 5 we see Case A results have their accuracy higher but their F1 score lower. This indicates a class imbalance because we are testing the species with test data proportional to their training set. Therefore there are more test cases for Yeast than E. coli. Which makes Yeast more dominant in the overall accuracy result. Since E. coli’s training set was smaller, the network was also more overfit on
F1 score takes into account precision and recall and is more informative in cases of class imbalance. F1 scores clearly show that the Case B is significantly better at classification. That is why we have opted to use the same number of samples for both species.

As we previously stated we took Baker’s Yeast as our positive outcome. During the tests we performed, F1 Score was always lower than the Accuracy Score. This can lead us to believe that the negative outcome which means classifying a read as E. coli is more likely. Since we performed these tests using Case B; E. coli is more represented in the training and test set which could cause this. Another reason might be due to the simplicity of the E. coli genome. This can be further investigated by comparing different species with genome simplicity in mind.

5.3 Using Synthetic Short Reads vs. Real-life Sequencing Outputs

We have tested our network on synthetic short reads taken from the reference genomes of the selected species. This was because our initial goal was fast classification of a long genome by taking multiple short reads and taking an average of the results we got. But more tests can be performed on real-life short reads to see how the network performs in such cases. Another option is to introduce base-flipping noise as elaborated on (Busia et al., 2019)[28].

Upon using such real-life sequences, the following topics should be considered:
• **Reads of Different Sizes**

Most sequencers do not produce fixed length results. The lengths of the reads produced can vary. Due to the nature of non-recurrent neural networks, the read length of the input on this network is fixed. However, there is a fifth category in the one-hot encoding that can be used in such an occasion. All the reads can be aligned to the left and padded with the character "N". In such a case, the model would need to be trained on such padded reads in the first place, or else it wouldn’t know how to handle them.

• **Possibly Mutated Sequences**

All genetic material mutates in some way or another. This is not reflected on this study. Reads that are not exactly present in the trained set can affect the performance of the network. It will however, classify the read as the species it finds it to be closer to. Therefore especially SNPs should not propose a big challenge for this network.

• **Assay Contamination**

Due to the small nature of the DNA, there is always a possibility of a contamination in the assay. Since we are training our model on labeled data, if the label is incorrect, it could worsen the performance of the model.

• **Sequencing Errors**

The error rates of NGS technologies have been shown to be around 0.1% [36]. This can obviously affect the performance of the model.

5.4 **Specifications of the Output Layer**

We have two nodes on our output layer, one for each species the network has trained on. We feed the desired outputs the same way we do our one-hot encoded inputs. [1, 0] for first class (Yeast), [0, 1] for second class (E. coli). Another option we had was to use one layer for binary classification [0] for yeast, [1] for e. coli. But using one node for each species gives us the opportunity to measure how confidently the network has classified a certain read.

We have used ReLU activation function on the output layer since it provided us with the best results. ReLU function maps our tanh function’s \([-1, 1]\) range into \([0, \text{inf})\). Since we have two species that do not have much in common genomewise, almost all of the results provide very similarly symmetric results when tanh function is used on the output layer (eg. 0.91 and -0.89). So even when using ReLU function, we can take the positive output as our confidence score. This would of course not be the case on multi-species classification and possibly not the case on binary classification with more common regions.

The linear nature of ReLU function may result in some outcomes where the comparison between certain outputs may not make as much sense.
5.5 Multi-species Classification

The network can be extended to go beyond binary classification and be trained on and classify multiple species data. The loss would need to become categorical cross-entropy, the number of nodes on the output layer would need to be adjusted to the number of species in the data.

The classification results’ metrics would need to be adjusted to accommodate multiple classes. Accuracy and F1 scores exist also for multi-species classification.

If more similar species are selected to classify, or more species are added to the model, the outputs will need to be closely monitored since more than one positive/negative outputs may be present. In such a case the best scoring classification may be taken or any classification that pass a certain threshold may be shown.

5.6 Comparison to similar work

Rizzo et al.[27] use randomly selected 1000 sequences from each of the three most common bacteria phyla, Actinobacteria, Firmicutes, Proteobacteria, collecting in total 3000 sequences. All the sequences have length greater than 1200 bp. They all belong to 16S genes. We are using all parts of the DNA for two species. They use sequences that are 500 bases long and also experiment with the full-length sequences of the genes. We also use 500 base long sequences but have also experimented with 50 base long sequences.

They use spectral sequence representations of the sequences. They extract k-mers where \( k = 5 \) from the sequence and list out their frequencies. They use this frequency data to train the model and perform the classifications on the test sets. This approach disregards any long distance correlations the sequence may hold. The frequency data is fixed in size which is important since they are using convolutional layers in their network. They have two convolutional and pooling layers followed by a hidden layer. We use the one-hot encoded genome sequences to train and test. We have three hidden layers.

They use tanh activation function throughout the network. We also use tanh function but on the output layer we switch to ReLU.

They are classifying the sequences down to the genus level. We are only interested in the species level of the classification. But since we have only two species that separate at the domain level, it could be said that we are classifying our sequences at the domain level.

Using 500 base long sequences, they achieve \( \sim 100\% \) accuracy on phylum and class level, \( \sim 90\% \) on order level, \( \sim 75\% \) on family level and \( \sim 50\% \) on genus level. We achieve \( \sim 91\% \) accuracy on domain level.

Busia et al.[28] use 16S rRNA sequences. They have 19,851 sequences with 13,838 from 2,768 genera. They use read lengths \( L = 25, 50, 100, 150, 200 \). We use short reads taken from the whole genome that have read length 50 to 500. We use two species.
To test their network, they used 16S mock community sequencing data and also synthetic data they simulated base-flipping noise on. We used synthetic small reads taken from the reference genome to train and test our network. They used one-hot encoding to encode the data similar to us.

They use three convolutional layers followed by two to three fully connected layers depending on the read length. They use an extra average pooling layer afterwards and use softmax activation function on the output layer. We use three hidden layers.

Weitschek et al.\cite{29} use DNA barcodes (COI for animals, rbcL and matK for plants). They require the sequences to be of the same region or at least to be pre-aligned so that the barcode regions can be extracted. We use short reads taken from the whole genome.

Since it is learning its rules on specific positions of the data, it requires a complete reference library of polymorphisms for each species in the training set to avoid false negatives. They achieve 92% accuracy on fungi and 100% accuracy on algae. We achieve 91% accuracy when classifying at the domain level.

Yang et al.\cite{30} use DNA barcode sequences to classify species. They use COI for animals, trnD-trnT and ITS for plants with 621 species in total. We use short reads taken from the whole reference genome for two species.

They use a convolutional network, treating one-hot encoding as its own dimension and convoluting and pooling in 2D. They then take the flattened pooling results through hidden layers, using ReLU as its general activation function and softmax in the last layer to compute the probabilities of the species. We have three hidden layers and use tanh activation function with ReLU function on the output layer.

5.7 Conclusion

Using DNA barcodes to classify DNA sequences is a common practice today. This method is lacking in situations where the sequence might not be accessible as a whole. Since most second generation sequencing machines tend to output reads of around 75 - 400 bp long, we need to be able to classify sequences of this length.

In this study, we set out to develop a deep learning method that could be trained on short reads taken from two reference genomes that can then successfully identify the origin of any short read taken from these reference genomes.

We have selected the two species to be Escherichia Coli and Saccharomyces Cerevisiae, due to their short and well-studied reference genomes. These two species are taxonomically very different, hence lack many identical DNA regions. This makes them ideal candidates to test the viability of using the whole genome for species classification.

We trained the network on short reads we randomly sampled from the reference genomes. The lengths of the reads were 50 to 500 for different networks. We performed our tests with same length reads. The model can accommodate the use of variable length reads. The nature of ANNs do not allow for variable length input but a recurrent layer can be introduced to combat this issue. Also the variable length reads can be padded with the character ’N’ to be made the same length. If variable length test cases will be encountered, the training set should also contain similar padded reads.
We tried different specifications for the neural network and achieved $\sim 83\%$ accuracy when predicting the species of short reads of 50 bases long. As we increased the read length up to 500 bases, we achieved $\sim 91\%$ accuracy.

5.8 Future Work

The inclusion of different deep learning techniques (RNNs, CNNs, transformers) could capture sequential correlations a simple ANN could not and might improve accuracy further.

The network can be extended to classify more than two species.

Finally, once reliable multiple species classification is achieved, further methods can be used to compress the data using the result of the classification. An autoencoder can be used to compress the data using the reference genome that belongs to that species. The compressed data would consist of the model, weights corresponding to the model and the difference the decompression of the compressed data would have with the reference genome.
REFERENCES


APPENDIX A

CONTENTS OF FILE PREP.PY

```python
import numpy as np
from pickle import dump
from sys import argv
from os import stat
from math import ceil
import tensorflow as tf
from tensorflow import keras
from numpy.random import random_sample

seed = 33333
keras.utils.set_random_seed(seed)
tf.config.experimental.enable_op_determinism()

def prepare_string_data_into_encoded_1D(string_data):
    result = np.zeros([len(string_data), 5], dtype=np.dtype(np.int8))
    for i in range(len(string_data)):
        if string_data[i] == 'A':
            result[i][0] = 1
        elif string_data[i] == 'T':
            result[i][1] = 1
        elif string_data[i] == 'C':
            result[i][2] = 1
        elif string_data[i] == 'G':
            result[i][3] = 1
        else:
            result[i][4] = 1
    return result

read_len = int(argv[1])
sample_representation = int(argv[2])
is_working_on_small_sample = int(argv[3])
species_length_equalized = 1
data_list = [['Bakers Yeast', 'Ecoli'], ['yeast', 'ecoli'], [0, 0], []]
data = []
len_samples_list = len(data_list[0])

open("pkl/" + str(read_len) + is_working_on_small_sample + "small" + ".pkl", "wb")
open("pkl/" + str(read_len) + is_working_on_small_sample + "small" + ".y.pkl", "wb")
data_list[2] = [0] * len(data_list[0])
```

```python
data_lens = [0 for i in range(len(data_list[0]))]
for i in range(len(data_list[0])):
    data = .join(np.genfromtxt(data/ + data_list[0][i] +
        is working on small sample small + .fna, delimiter=n, dtype = str)).upper()
    data = prepare_string data into encoded 1D(data)
data_lens[i] = len(data)
max data = max(data_lens)

for i in range(len(data_list[0])):
    data = .join(np.genfromtxt(data/ + data_list[0][i] +
        is working on small sample small + .fna, delimiter=n, dtype = str)).upper()
    data = prepare_string data into encoded 1D(data)
if(species length equalized):
    samples indices = ((len(data) read len) random sample(size = int(
        max data sample representation/read len))).astype(int)
else:
    samples indices = ((len(data) read len) random sample(size = int(
        len(data) sample representation/read len))).astype(int)
samples = np.empty([len(samples indices), read len, 5], dtype = np.dtype(
    np.int8))
for j in range(len(samples indices)):
    index = samples indices[j]
    sample = data[index:index+read len]
    samples[j] = sample
len samples list[i] = len(samples)

with open("pkl/" + str(read len) + is working on small sample "small"
    + ".pkl","ab") as f:
    dump( samples, f )
data_list[2][i] = len(samples)
y = np.asarray([np.array([0] + [1] + (len(data_list[0]) i l) [0]) for
    j in range(data_list[2][l])]), dtype = np.dtype(np.int8))
with open("pkl/" + str(read len) + is working on small sample "small"
    + ".pkl","ab") as f:
    dump( y, f )
with open("pkl/data list " + str(read len) + is working on small sample "small"
    + ".pkl","wb") as f:
    dump( data_list, f )
```

40
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras.callbacks import LambdaCallback
from keras.layers import Flatten, Dense, Activation
from keras import activations, losses, initializers, metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, recall_score, precision_score
from sys import import
from pickle import load
from time import time
from gc import collect
from math import ceil

tf.config.experimental.enable_op_determinism()
seed = 0
batch_size = 128
std_dev = 0.03 0.09 for 50, 0.03 for 500
mean = 0.0
learning_rate = 0.00003 0.0006 for 50, 0.00003 for 500
keras.utils.set_random_seed(seed)
epochs = 5

print("seed": seed, ", start time": time().end =", ")
is working on small sample = int(argv[1])
read_len = 500 int(argv[2])
encoding_dim_layer_1 = 1000 200 for 50, 1000 for 500
encoding_dim_layer_2 = 100 100 for both 50 and 500
encoding_dim_layer_3 = 20 20 for both 50 and 500
activation_function = activations.tanh
loss = losses.BinaryCrossentropy(from_logits=True)
sample_representation = 2
one_hot_encoding_size = 5

with open("pkl/data list" + str(read_len) + is working on small sample "small" + ".pkl", "rb") as f:
data_list = load(f)
for i in data_list[2]:
  print(i, end = ", ")

initializer = initializers.RandomNormal(stddev=std_dev, mean=mean)
initializer name = "random normal"
layers
flatten = Flatten()
dense1 = Dense(encoding_dim layer 1, kernel initializer=initializer)
activation1 = Activation(activation function)
dense2 = Dense(encoding_dim layer 2, kernel initializer=initializer)
activation2 = Activation(activation function)
dense3 = Dense(encoding_dim layer 3, kernel initializer=initializer)
activation3 = Activation(activation function)
output = Dense(len(data list[0]), kernel initializer=initializer)
activation4 = Activation(activations.relu)
data flow
input = keras.Input(shape=(read len, one hot encoding size))
flattened = flatten(input)
encoded1 = dense1(flattened)
activated1 = activation1(encoded1)
encoded2 = dense2(activated1)
activated2 = activation2(encoded2)
encoded3 = dense3(activated2)
activated3 = activation3(encoded3)
encoded triangle = output(activated3)
activated triangle = activation4(encoded triangle)
triangle = keras.Model(input, activated triangle)

x test, y test = np.array([]), np.array([])
len x = sum(data list[2])
X full = np.zeros([len x, read len, one hot encoding size], dtype=np.dtype(np.int8))
it = 0
filename = "pk1/" + str(read len) + is working on small sample "small" + ".pk1"
with open(filename,"rb") as f:
  for i in range(len(data list[0])):
    sample = load(f)
    for j in sample:
      X full[it] = j
    it += 1
del sample
collect()

Y full = np.zeros([len x, len(data list[0])], dtype=np.dtype(np.int8))
it = 0
y filename = "pk1/" + str(read len) + is working on small sample "small" + "+ y.pkl"
with open(y filename,"rb") as f:
  for i in range(len(data list[0])):
    sample = load(f)
    for j in sample:
      Y full[it] = j
    it += 1

start time = time()

x train, x test, y train, y test = train test split(X full, Y full, random state=seed, test size =0.20) 80 train
x val, x test, y val, y test = train test split(x test, y test, random state=seed, test size =0.5) 10 val 10 test

42
96 del X_full, Y_full
97
98 printerCallback = LambdaCallback(
99     on_epoch_begin = lambda epoch , logs: print(epoch, end = ""),
100     on_epoch_end = lambda epoch , logs:
101        print(":::.4 f \ .format(\[ loss ]), " , " :::.4 f \ .format(\[ val loss ]), " , " :::.1f sec".format(ceil((time()
102         train_start_time) 10)/10), end = "")
103 )
104 opt = tf.keras.optimizers.Adam(learning rate)
105
106 y_true = [0 if x[0] x[1] else 1 for x in y_test]
107 temp = triangle.predict(x_test, verbose=0)
108 avgs = np.average(temp, axis=0)
109 print("untrained output averages: " , "[ :::.5 f , :::.5 f ]".format(avgs[0], avgs
110     [1]), end = "")
111 train_start_time = time()
112 print(" sample representation , " , " read len , " , " str(loss) . split( ")
113     [1] , " , " , encoding dim layer 1 , " , " , encoding dim layer 2 , " , " ,
114     encoding dim layer 3 , " , " , str(activation function) . split( ")[1] , " , " ,
115     epochs , " , " , train start time , end = ", " , )
116 triangle.fit(x_train, y_train,
117     epochs=epochs,
118     batch size=batch size,
119     shuffle=True,
120     validation data=(x_val, y_val),
121     callbacks=[printerCallback],
122     verbose=0
123 )
124 temp = triangle.predict(x_test, verbose=0)
125
126 y_pred = [0 if x[0] x[1] else 1 for x in temp]
127 y_true = [0 if x[0] x[1] else 1 for x in y_test]
128 with open("output pred real weights . csv", "w") as f:
129     for i in range(len(temp)):
130         f . write(":::.2 f , :::.2 f , , n".format(temp[i][0], temp[i][1],
131                     y_pred[i], y_true[i]))
132 print("end time:" , time().end = "")
133 print("Accuracy: ", accuracy score(y true , y pred),
134       ", Recall: ", recall score(y true , y pred , average="binary"),
135       ", Precision: ", precision score(y true , y pred , average="binary"),
136       flush=True)
137 keras.backend.clear session()
# APPENDIX C

## LIST OF INSTALLED PACKAGES AND VERSIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>_tflow_select</td>
<td>2.1.0</td>
</tr>
<tr>
<td>cudatoolkit[37]</td>
<td>11.3.1</td>
</tr>
<tr>
<td>cudnn</td>
<td>8.2.1</td>
</tr>
<tr>
<td>ipykernel</td>
<td>6.13.0</td>
</tr>
<tr>
<td>ipython</td>
<td>8.2.0</td>
</tr>
<tr>
<td>ipython_genutils</td>
<td>0.2.0</td>
</tr>
<tr>
<td>jupyter_client</td>
<td>7.3.0</td>
</tr>
<tr>
<td>jupyter_core</td>
<td>4.9.2</td>
</tr>
<tr>
<td>jupyter_server</td>
<td>1.17.0</td>
</tr>
<tr>
<td>jupyterlab</td>
<td>3.3.4</td>
</tr>
<tr>
<td>jupyterlab_p gypsum</td>
<td>0.2.2</td>
</tr>
<tr>
<td>jupyterlab_server</td>
<td>2.13.0</td>
</tr>
<tr>
<td>keras[32]</td>
<td>2.9.0</td>
</tr>
<tr>
<td>keras-preprocessing</td>
<td>1.1.2</td>
</tr>
<tr>
<td>notebook</td>
<td>6.4.11</td>
</tr>
<tr>
<td>notebook-shim</td>
<td>0.1.0</td>
</tr>
<tr>
<td>numpy</td>
<td>1.23.4</td>
</tr>
<tr>
<td>pdflatex</td>
<td>0.1.3</td>
</tr>
<tr>
<td>pickleshare[34]</td>
<td>0.7.5</td>
</tr>
<tr>
<td>pip</td>
<td>23.2.1</td>
</tr>
<tr>
<td>python[34]</td>
<td>3.9.12</td>
</tr>
<tr>
<td>rsa</td>
<td>4.7.2</td>
</tr>
<tr>
<td>scikitlearn[38]</td>
<td>1.1.1</td>
</tr>
<tr>
<td>scipy</td>
<td>1.9.3</td>
</tr>
<tr>
<td>sklearn</td>
<td>0.0</td>
</tr>
<tr>
<td>tensorflow[33]</td>
<td>2.9.1</td>
</tr>
<tr>
<td>tensorflow-data-server</td>
<td>0.6.0</td>
</tr>
<tr>
<td>tensorflow-plugin-wit</td>
<td>1.6.0</td>
</tr>
<tr>
<td>tensorflow-estimator</td>
<td>2.9.0</td>
</tr>
<tr>
<td>tensorflow-gpu[33]</td>
<td>2.6.0</td>
</tr>
<tr>
<td>tensorflow-io-gcs-filesystem</td>
<td>0.26.0</td>
</tr>
</tbody>
</table>

Table 7: Packages in environment tf_gpu