A Recurrent Neural Network approach for whole genome bacteria classification

Luis Eduardo Lugo Martínez

National University of Colombia
Faculty of Engineering
Department of Systems and Industrial Engineering
Bogotá, Colombia
2018

A Recurrent Neural Network approach for whole genome bacteria classification

Luis Eduardo Lugo Martínez

A final project submitted in partial fulfillment of the requirements for the degree of:
Master in Systems and Computer Engineering

Advisor:
Ph.D. Emiliano Barreto Hernández

Line of Research:
Bioinformatics and Health
Research Group:
Bioinformatics

National University of Colombia
Faculty of Engineering
Department of Systems and Industrial Engineering
Bogotá, Colombia
2018

To my family

Acknowledgements

I would like to thank the National University of Colombia for providing an amazing learning environment that stimulates education, scientific research, and cultural development.

I would also like to thank Professor Ph.D. Emiliano Barreto for his guidance throughout a fascinating field of research. Also, his support and advice were fundamental to the successful completion of the project.

Last but not least, I would like to thank my family for their unconditional support, encouragement, and understanding during the whole process.

Abstract

The classification of bacteria plays an essential role in multiple areas of research. Those areas include experimental biology, food and water industries, pathology, microbiology, and evolutionary studies. Although there exist methodologies for classification – such as mass spectrometry, single-nucleotide polymorphisms, microscopic morphology, and neural network approaches – a transition to a whole genome sequence based taxonomy is already undergoing. Next Generation Sequencing helps the transition by producing DNA sequence data efficiently. However, the rate of DNA sequence data generation and the high dimensionality of such data need faster computer methodologies.

Machine learning, an area of artificial intelligence, has the ability to analyze high dimensional data in a systematic, fast, and efficient way. Therefore, we propose a sequential deep learning model for bacteria classification. The proposed neural network exploits the vast amounts of information generated by Next Generation Sequencing, in order to extract a classification model for whole genome bacteria sequences. A distributed representation based on k-mers of $k = \{3, 4, 5\}$ provided an efficient encoding for the bacterial sequences. The classification model relies on a bidirectional recurrent neural network architecture. It generates an accuracy of 0.99455 ± 0.00281 for 14 species, 0.95031 ± 0.00469 for 48 species, and 0.89107 ± 0.00392 for 111 species. After validating the classification model, the bidirectional recurrent neural network outperformed other classification approaches, such as Naive Bayes and Feedforward neural network. The proposed model provides an automated identification method. It infers species for bacterial whole genome sequences and it does not require any manual feature extraction.

Keywords: whole genome sequence, recurrent neural network, deep learning, bacteria classification

Contents

List of Figures

List of Tables

Chapter 1
Introduction

A principle is merely a statement of an impossibility

Alexandre Koyré
(Desolneux et al., 2007)

In biology research, it is fundamental to have classification methods to properly identify all the organisms. Classifications methods based on phenotype have considerable flaws, including the need for cultured samples and the lack of differentiation among certain groups (Olive and Bean, 1999; Mohamad et al., 2014). Therefore, classification based on genotype or DNA sequence analysis is a better approach for biological identification (Olive and Bean, 1999).

Along archaea and eukaryota, bacteria constitute one of the three major branches in the evolutionary tree. Microbes are relatively simpler than other organisms -although the simplicity is often deceptive. Because of this, they help to answer very important questions in science. Among microbes, bacteria are highly adaptable unicellular organisms. They can usually survive and thrive in extreme environments. Moreover, bacteria are structurally simpler and haploid. So, they serve as model organisms to study gene mutations and their specific function. These factors make them a platform to study complex eukaryotic organisms, thus, bacteria have been exploited to perform vast amounts of genetic analysis (Srivastava, 2016; Lodish et al., 2004).

In particular, the identification of bacteria plays an important role in multiple areas of research. Experimental research in biology benefits from the elegant mechanisms of gene activity control in bacteria, the rapid rate at which they grow, and the powerful genetics they have (Lodish et al., 2004). Pathology and microbiology also benefit from bacteria taxonomy as it serves as the basis for understanding certain diseases (Mohamad et al., 2014). Moreover, a fast identification of bacteria is critical to ensure the quality of water and food products (Singhal et al., 2015). It is also important from the evolutionary standpoint because it allows the documentation of changes in genes and proteins (Lodish et al., 2004).

Starting from the taxonomy system in (Woese et al., 1990), a very commonly used method for bacteria classification is based on gene sequences comparisons of small ribosomal subunits. This method is known as 16S rRNA-based taxonomy. However, recent advances in next-generation sequencing (NGS) – and the availability of algorithms and heuristics for DNA sequence analysis – are easing the transition to whole genome sequence (WGS) taxonomy.

Indeed, WGS methods are taking over bacteria taxonomy, which is defined as the process of systematic classification of species and strains. But it is far from being an official classification. Instead, it is a flexible classification that can change according to new advancements in biology. Names and description do not constitute facts. They are susceptible to revisions and opinionated changes. The importance of WGS methodologies is related to the speed of processing and the increase of taxonomy resolution (Garrity, 2016). The latter is considered one of the most significant drawbacks of 16S mRNA-based technologies. The increased resolution is quite relevant as classification methods should be able to differentiate an increasingly larger number of organisms

Recasting existing taxonomy to new classification data from WGS is already happening. However, 16S rRNA methodologies will continue to be an important area because they are the benchmark used for comparison of emerging methodologies. They enabled the creation of a base map for taxonomy data – which has unparalleled importance in life science research (Garrity, 2016).

The most common method for WGS classification is the average nucleotide identity (ANI). New algorithms have been developed to implement this method and they are now routinely used. ANI compares two sequences with more than 70% similarity and generates a similarity index. Based on that index, the relationship between sequences of the same species is considered in the index range of 94% to 96%. Nonetheless, this kind of method is computationally expensive because sequences must be aligned beforehand. Also, the comparison is conducted between pairs at a time, which takes considerable resources when dealing with high-throughput sequence analysis (Varghese et al., 2015; Garrity, 2016).

Thus, as advances in NGS generate high dimensional biological data at a faster rate, and more curated databases of biological sequences are publicly available, there exists the need to process considerable amounts of high dimensional data in a systematic and efficient way. The unprecedented amount of genomic data generated by NGS platforms increases the demand for large-scale data analysis (Pais et al., 2014). Extracting useful information from those large datasets is challenging. Manual processing of such volumes of data is slow, expensive, and impractical. It is also error prone and highly subjective (Fayyad et al., 1996). Often, traditional software tools can not process the large datasets or efficient processing algorithms can not be developed beforehand (Dahl, 2015; Tan et al., 2018). In fact, available software systems and algorithms for biological sequence processing should be improved. Computational tasks pose some of the biggest challenges in processing efficiency regarding WGS projects (Scholz et al., 2012).

In the traditional approach for software development, practitioners design a set of steps and then program a computer to execute them. Depending on the input data, the computer executes the set of steps and generates the corresponding results. However, it is widely accepted that biological systems start from a basic set of rules, for instance, random variation and natural selection. Building on top of those simple rules, they allow the evolution and adaptation of organisms to changing environmental conditions. Inspired by such approach, computer scientists stopped focusing only on building systems using a top-down method. Instead of programming complex rules to fully specify a computing system, they started to take a bottom-up approach. First, they program a basic set of rules and then, they let the system adapt through multiple iterations and automatically infer their own complex sets of rules (Dahl, 2015; Mitchell, 1995).

One of the methods that leverages the bottom-up approach is machine learning. It provides what is known as a ”meta-algorithm”, which enables practitioners to tell the computer what needs to be solved, instead of telling it how to solve it (Dahl, 2015; Mitchell, 1995). Because of this, machine learning enables a set of methods to analyze high dimensional data and extract patterns from it. The results might be used to predict future events through models or make informed decisions when different possibilities create uncertainty. Probability theory is the base for machine learning methodologies, as that area of mathematics is the natural choice for dealing with uncertainty (Murphy, 2012).

Deep neural networks are considered a turning point in machine learning, setting records in some tasks that surpass human capabilities (Pastur-Romay et al., 2016). They are particularly tailored to extract patterns from high-dimensional data, which make them applicable to different areas of science, business, and government. Such type of neural networks has been giving impressive results as of lately, even surpassing other machine learning methods in diverse areas such as speech recognition, visual object recognition, natural language understanding, particle accelerator data analysis, prediction of potential drug molecules activity, and effects on gene expression and disease caused by mutations in noncoding DNA (LeCun et al., 2015; Dahl, 2015).

In fact, there is an explosion of deep learning applications for research in bioinformatics (Min et al., 2016; Nguyen et al., 2016). Both bioinformatics and computational biology are taking the benefits of deep neural network approaches to leverage very large datasets of high dimensional data. Biological datasets are explored to extract hidden structure within them and to make accurate predictions. Applications of deep neural networks are found in regulatory genomics, drug discovery, cellular imaging, medical imaging, genomic data mining, biomedical signal processing, and sequence classification (Webb, 2018; Min et al., 2016; Nguyen et al., 2016; Angermueller et al., 2016; Bosco and Di Gangi, 2016). In particular, sequence classification is challenging because sequences do not have explicit features. The feature extraction methods require domain expert knowledge and are highly problem–specific. These methods can be labor intensive as well, thus representing a bottleneck for high-dimensional genomic manual data processing (Angermueller et al., 2016).

1.1 Contribution and outline

The main contribution of the project is to understand how the application of deep neural networks can improve existing DNA sequence analysis tools for Whole Genome Sequence classification of bacteria. An improved identification system has the potential to help the ongoing transition from 16s ribosomal RNA taxonomy to a whole genome based bacteria taxonomy. Also, it takes advantage of the increasing amount of data generated from Next Generation Sequencing technologies. This is particularly important because other methods become computationally more expensive as more data is available. But deep learning methods are data-intensive applications that benefit from an increasing amount of data available.

To achieve that goal, we propose a sequential deep learning architecture – a recurrent neural network – for bacteria classification, using next generation sequencing data. There are four specific objectives: prepare the NGS bacterial data from GenBank, find the genome sequence representation that best suits a sequential deep neural network, implement the recurrent neural network architecture that best processes whole genome sequences, and validate results with an existing model for bacteria classification.

The remaining of the document is organized as follows: Chapter 2 includes a review comprising bacteria classification methods and deep neural networks. Chapter 3 focuses on the method used for bacteria classification with recurrent neural networks. Experimental setup, results, and validation appear in chapter 4. Finally, concluding remarks and future research are discussed in chapter 5.

Chapter 2
Bacteria Classification

Bacteria have several advantages as experimental organisms: They grow rapidly, possess elegant mechanisms for controlling gene activity, and have powerful genetics

(Lodish et al., 2004)

There exists a number of criteria for the classification of bacteria and archaea, including morphology, genome size, lifestyle, relevance to human disease, molecular phylogeny using rRNA, and genomic sequence analysis (Pevsner, 2015; Mohamad et al., 2014). We consider different bacteria identification methods: mass spectrometry (Singhal et al., 2015), whole genome sequencing for clinical samples (Hasman et al., 2013), genome wide Average Nucleotide Identity (Varghese et al., 2015; Garrity, 2016), pattern recognition in image processing (Mohamad et al., 2014), and deep neural network architectures (Rizzo et al., 2015; Bosco and Di Gangi, 2016).

We discuss existing approaches for bacteria identification, having in mind important aspects such as the methodology used by the alternative, the dataset used for the classification system or the group of bacteria considered for the identification, the number of ordered taxonomic ranks, if any, the results obtained and reported quantities – which could be accuracy, precision, recall, identity percentage, and so on – and the limitations of the method.

2.1 Mass spectrometry

Mass spectrometry is a powerful mechanism devised to measure the mass of molecules. In biology, such molecules include proteins and peptides. The technique ionizes the molecules using laser energy. Then, using electric fields, the ions are accelerated towards a detector. The time of flight is proportional to the charge of the ion. But it is inversely proportional to the mass of the molecule, allowing detectors to discriminate them (Singhal et al., 2015; Lodish et al., 2004).

Matrix-assisted laser desorption ionization-time of flight mass spectrometry (MALDI-TOF MS) is one of the approaches using the mass spectrometry mechanism. It is a fast, sensitive, and cost-efficient methodology for microbial identification. It has been used by microbiologists for identification of diseases caused not only by bacteria, but fungi and viruses as well. Other areas leveraging this technology include the detection of food- and water-borne pathogens, identification of biological warfare agents, epidemiology, microbial identification, and strain typing (Singhal et al., 2015).

Advantages include the realization of tests with cells or extract of cells. Also, the method is accurate and less expensive than other methods based on molecular detection. Among mass spectrometry limitations, it is necessary to have entries in the database containing the results of peptides related to the type strain of the organism. This should happen to correctly perform the identification at the genera, species, subspecies, and strain level. Moreover, mass spectrometry requires costly laboratory equipment (Singhal et al., 2015).

2.2 Whole genome sequencing for clinical samples

In clinical microbiology, a rapid extraction of relevant information from clinical samples helps the speed of diagnosis, positively impacting control and treatment. Whole genome sequencing has promising applications in public health. It improves the understanding of bacterial evolution, outbreaks, and transmission. It also helps to understand and potentially limit the inter-hospital proliferation of pathogens. By the same token, WGS applied over the clinical samples has the potential to considerably improve the diagnostic times. However, there is a need for fast and reliable bioinformatics tools for the data analysis required when processing whole genome sequencing results (Hasman et al., 2013).

Chainmapper.py is a bioinformatics tool developed to address this need. To conduct the research, Hasman et al. (2013) randomly choose 35 urine samples from patients suspected to have urinary tract infections (UTIs). Urine samples are less complex than other clinical examples because they have low contamination from human DNA and high concentration of bacterial cells. The samples are analyzed by using three methods: conventional microbiology diagnostic methods, whole genome sequencing of bacterial isolated samples, and direct sequencing of pellets. Conventional microbiology methods use agar plates to cultivate the samples, then it performs susceptibility tests and finally, it characterizes each sample. This procedure can take several days. Bacterial isolated samples require a few days for culturing and subsequent identification. Direct sequencing is the fastest of the three methods, requiring less than 24 hours.

The Ion Torrent PGM system provides the sequencing for the bacterial isolates. Once the sequencing results are obtained, a 16-mers based method provides identification against complete bacterial genomes from NCBI. MG-RAST[1], a metagenomics analysis server (Meyer et al., 2008), enables the calculation of host contamination and the proportion of bacterial species present in the sequencing results. Chainmapper.py is the primary tool for the analysis. It performs species identification. But it also calculates an abundance estimation graph and a microbial community profile. The profile is computed by aligning the sequencing reads to a number of genomes, covering a human genome, complete and draft bacterial genomes, complete and draft fungal genomes, and complete and draft protozoan and viral genomes (Hasman et al., 2013).

An online database with almost 2000 variants provides information about resistant genes in the samples (Zankari et al., 2012). Multilocus sequence typing information is also included. The snpTree[2] web server (Leekitcharoenphon et al., 2012) computes the phylogenetic maps with the most common bacterial species.

Results from direct sequencing of samples - the fastest of the three methods - are highly reliable when compared to isolated bacterial samples, giving the same results in less time. Also, result analysis enabled the identification of bacterial species that were not identified by conventional microbiology diagnostic methods (Hasman et al., 2013). On the other hand, disadvantages of the k-mers based identification method include the need to perform a database search for every sample, which can be time-consuming depending on the number of sequences. Also, Chainmapper.py is computationally expensive, requiring a high-performance computing setup to provide fast results.

2.3 Genome wide average nucleotide identity

The polyphasic approach is a traditional method for asexual organism classification. It relies on genotypic, phenotypic, and chemotaxonomic data in order

[1] http://metagenomics.anl.gov

[2] https://cge.cbs.dtu.dk/services/snpTree/

to identify organisms. The species delineation is then performed on a consensus of existing information. However, the whole-genome information of an organism is its ultimate genetic signature. Also, sequencing whole genomes is fast, accurate, and affordable nowadays. Thus, a better approach is to compute genomic distance on whole genomes for identifying microbial species (Varghese et al., 2015). To implement this approach, (Varghese et al., 2015) propose the Microbial Species Identifier (MiSI). They also propose its use to correct inconsistencies in existing taxonomic data and as a guide for new species assignments –supplementing it with the polyphasic approach when the need arises.

MiSI is a method to define prokaryotic species based on a combination of two metrics: alignment fraction (AF) and genome-wide average nucleotide identity (gANI). AF is the fraction of orthologous genes between a pair of sequences. The code for AF and gANI computation is freely available at https://ani.jgi-psf.org/html/download.php. It uses the MSimScan [3] tool for comparing pairs of genomes. After computing the AF and gANI values for the pairs of genomes, a clustering process creates an undirected graph relating whole genomes. Vertices of the graph are genomes while edges connect genomes with AF greater than 0.6, and gANI greater than 96.5.

Genomic data for this study was part of the Integrated Microbial Genomes (IMG) database[4] (Markowitz et al., 2011), a publicly available dataset which also has tools and viewers for analysis of genomic information in a comparative context. Using the MiSI method, an inconsistency of around 18% was found in taxonomic species definitions. The testing phase comprised 13151 prokaryotic genomes, 85.5M pairs of genomes, and 3052 species.

As far as advantages of the MiSI method are concerned, the graph construction is independent of taxonomic data because it relies only on genomic information. Also, Varghese et al. (2015) found that a genome reduction of up to 25% had no effect on the results, making it reliable when draft genomes are present. On the other hand, pairwise comparisons take more computing resources than other automatic approaches. Also, as the number of sequences grows, the number of pairs increases exponentially, generating a larger graph. The approach also neglects physiological changes generated by variations in single genes or small subsets of genes.

2.4 Microscopic morphology

An approach that incorporates machine learning techniques as part of the processing pipeline is the microscopic morphology identification. It is an image based methodology that helps to identify bacteria in an automated way.

[3] http://www.scidm.org/

[4] https://img.jgi.doe.gov/

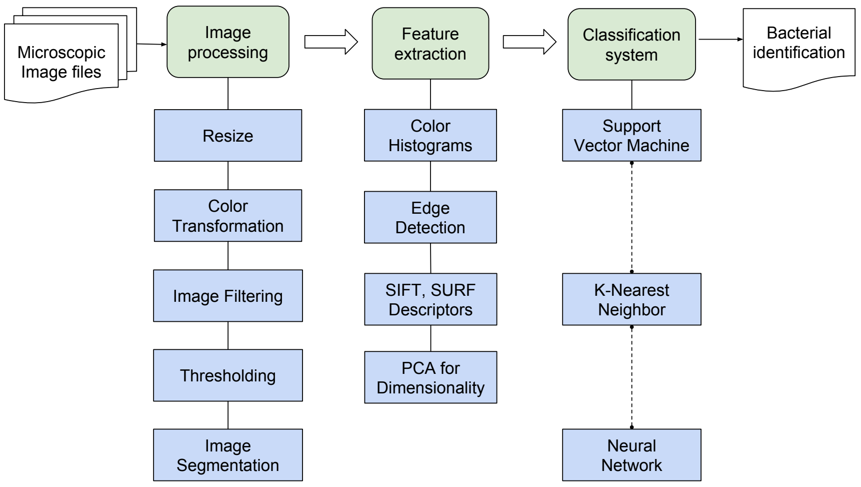


Fig. 2.1: Classification of bacteria based on microscopic morphology identification. The diagram depicts three phases for identification: image processing, feature extraction, and classification (Hassaballah et al., 2016; Mohamad et al., 2014).

It also facilitates the construction of predictive models that take advantage of morphology features and cell arrangements. The application of image processing algorithms to bacterial microscopic images enables the identification of organisms in high throughput analyses, which can process hundreds of images at a time (Mohamad et al., 2014).

The processing includes three phases: image processing, features extraction, and classification (Figures **??**). Image processing comprises resizing, segmentation, color transformations, filtering, or thresholding. Once the microscopic image is preprocessed, a second stage extract features using color histograms, edge detectors, or image descriptors (Mohamad et al., 2014). Popular descriptors include Speeded-up Robust Features (SURF), Scale Invariant Feature Transform (SIFT), and Fast Retina Keypoint (Hassaballah et al., 2016; Alahi et al., 2012). If the features suffer from high dimensionality, techniques such as principal component analysis (PCA) could be used for dimensionality reduction. After extracting features per image, a classification system identifies organisms using a number of methods such as support vector machines (SVMs), K-Nearest Neighbors, or neural networks (Mohamad et al., 2014; Murphy, 2012).

Advantages of morphological identification systems are their simplicity, fast processing, automatic classification, and low amount of human intervention. Disadvantages include the need to have curated datasets for training in the classifier stage. Also, the reliance on phenotypical information for identification has been criticized for the need of cultured samples and the lack of differentiation among certain groups (Olive and Bean, 1999; Mohamad et al., 2014).

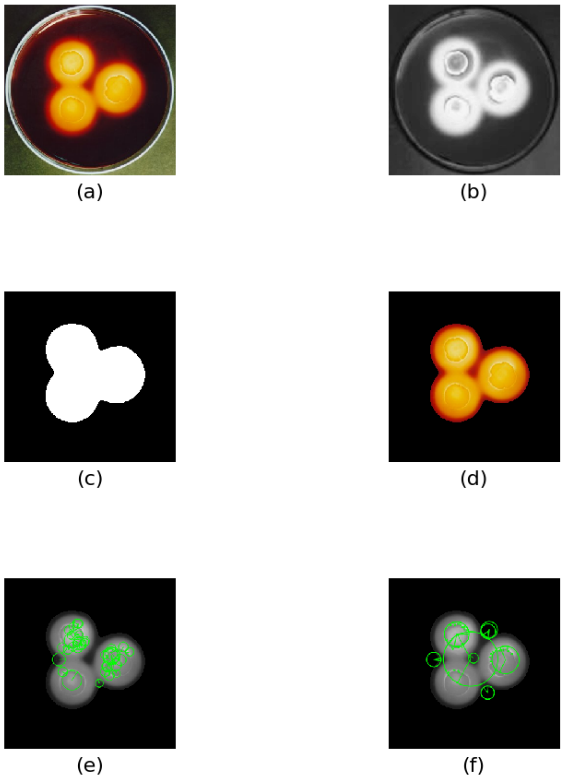


Fig. 2.2: Some image processing steps in the microscopic morphology identification pipeline. (a) Original microscopic image. (b) I_2 component of the OHTA color space. (c) Mask from filtering and thresholding the I_2 component. (d) Image segmenation. (e) Keypoints for FREAK descriptor. (f) Keypoints for SIFT descriptor. Original image from http://atlas.sund.ku.dk

2.5 Deep neural networks

Based on neuroscience research, artificial neural networks emulate the processing power of the large interconnected network of neurons inside our brain. Their purpose is to understand the computational theory behind the brain and the way it abstractly represents human intelligence. Artificial neural networks were originally created as mathematical models of the brain processing tasks. Although nowadays it is known that neurons in the brain work in a different manner, neural networks continue to provide popular pattern classifier models. The neural network comprises nodes – processing units – with activation functions interconnected in layers through weights. Nodes mimic the neurons, while the weight connections try to simulate synapses in the brain. To train the neural networks, existing data sets provide an input – output relationship that the network discovers through the learning process (Alpaydin, 2014; Graves, 2012).

Research in this area not only helps us approach problem solving using computing resources. It also contributes to the understanding of how the brain works, and how humans and animals learn. Similarly, brain research provides machine learning researchers with guidance to improve neural network architectures (Dahl, 2015). For instance, Glorot et al. (2011) take ideas from computational neuroscience to improve neural network models. They focus their analysis on deep neural networks with rectified linear units (ReLUs), achieving considerable improvements by proving the way ReLU based hidden layers make training much faster. Indeed, ReLUS are the most common implementation to model brain neuron activation. Neuroscience inspired the fact that simple units can give excellent results when they work in a hierarchical interconnected arrangement (Goodfellow et al., 2016).

Neural networks also allow for an efficient computation in parallel architectures, which are particularly difficult to program. Instead of manually programming parallel processors, neural networks can be trained to compute their own parameters. To train them, existing data sets provide an input-output relationship that the network discovers through the learning process (Alpaydin, 2014). This is known as supervised learning, where a set of inputs is mapped to a certain output, and the parameters of the system are changed to improve the matching of the model. Other learning methods include unsupervised learning and reinforcement learning. In unsupervised learning, there is a set of inputs but no expected output, thus, the model should extract patterns from the inputs to create valid parameters. Reinforcement learning is based on a system of punishments or rewards for the outputs generated, which tailor the system for appropriate modeling (Murphy, 2012).

Neural networks can be classified as shallow or deep. The main difference between shallow and deep neural networks is the length of the possibly learnable causal chains established throughout the processing layers (Schmidhuber, 2015). Therefore, deep neural networks receive their name from the number of layers between input and output stages. More precisely, the term deep refers to the path between input and output stages when considering the neural network as a direct path. It also refers to the ability to compose simpler units or representations to express complex concepts. The composition of simple mathematical units into networks can solve complex problems requiring intelligence (Goodfellow et al., 2016; Rizzo et al., 2015).

Traditional machine learning algorithms depend heavily on the way input data is represented. Each attribute of the proposed representation is known as a feature. It usually requires considerable time and effort to find the correct way to represent input data in order to get a good performance in the machine learning method. The process also needs high domain specific knowledge to extract the most important attributes of the input dataset. But automatically learned representations usually have a better performance than handcrafted feature extraction. When methods learn to represent input data by themselves, we are talking about representation learning. Deep learning models are a kind of representation learning. They learn not only the mapping between features and results, but the representation of input data itself (Goodfellow et al., 2016).

Therefore, deep neural networks provide a way to infer models from the raw data by extracting hidden structures from the information. They also automatically extract the set of attributes that best represents the input dataset – improving the learning process by encoding representations as a nested hierarchy of simpler or less abstract representations (Rampasek and Goldenberg, 2016; Goodfellow et al., 2016). For example, the Merck Molecular Activity Challenge 2012 was won by machine learning experts with no background in chemistry. The team led by George Dahl (Dahl, 2015) outperformed other techniques using only minimal preprocessing and no previous feature extraction.

Very similar deep neural network architectures have worked for various case studies in not related domains, which proves the suitability of deep learning models to adapt to various problem domains. At the same time, it proves the capacity of deep neural models to automatically extract features from data in spite of dataset domain origin. It also showcases the ability of deep neural networks to learn non–linear transformations in their hidden structures and to generate distributed representations of input datasets. The suitability of deep learning models for both supervised and unsupervised approaches are also clearly seen from the multiple domains in which this machine learning architecture is increasingly finding more applications (Dahl, 2015).

Disadvantages of deep learning architectures include the black box nature of their processing layers and the inability to correct their wrong answers. Currently, it is not possible to understand the inner processing performed by such architectures. Therefore, good models can not be reused or replicated. Moreover, when there are errors in input data processing, researchers are unable to repair the inner structure to get correct answers (Rizzo et al., 2015). Another disadvantage of deep neural networks is the training process, as some architectures are difficult to properly train (Srivastava et al., 2014). By the same token, the results from these architectures depend on the quality of input data. Thus, they require well annotated and large datasets for successfully training the models (Webb, 2018).

2.5.1 Common deep learning architectures

Nowadays, popular deep learning architectures comprise Multilayer Perceptrons, Convolutional Neural Networks, and Recurrent Neural Networks (Jouppi et al., 2017).

Multilayer perceptrons (MLPs) represent models of artificial neural networks that are useful for classification and regression tasks. The perceptron is the basic processing unit of the MLP (Figure 2.3). It comprises a series of weights that transform the input nodes through a mathematical operation. Then, an output node computes the result by combining such transformations into a single value. If a system uses only one perceptron, it can only approximate linear functions of input data. Thus, it only performs linear regression. But real applications usually require nonlinear regressions. MLPs implement hidden layers of perceptrons between input and output nodes. Those hidden layers enable the approximation of nonlinear functions for the input data. An MLP with a single hidden layer and enough nonlinear nodes – processing units – can approximate any continuous function in a compact domain. Therefore, they are considered universal function approximators. (Alpaydin, 2014; Graves, 2012).

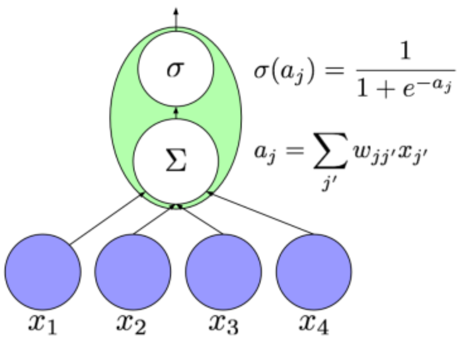

σ
Σ
$\sigma(a_j) = \frac{1}{1+e^{-a_j}}$
$a_j = \sum_{j'} w_{jj'} x_{j'}$
x_1
x_2
x_3
x_4

Fig. 2.3: A perceptron with sigmoid as the activation function. From (Lipton et al., 2015)

Both hidden layers and output nodes in MLPs have nonlinear activation functions. Once one or more input nodes are activated, the information propagates through the whole set of interconnected nodes in the subsequent layers, up to the output layer. Such propagation is known as a forward pass. The MLP response does not depend on the past or future inputs, it only depends on the current input. Because of this, they are a good alternative for pattern classification tasks (Graves, 2012).

Convolutional neural networks (CNNs) are models inspired by biological visual systems. Specifically, they are inspired by the visual cortex of the brain, where a combination of simple and complex neurons interact to build the powerful natural visual system. CNNs comprise convolution layers, nonlinear layers, and pooling layers. These models are quite successful nowadays for their ability to process spatial and multidimensional information (Min et al., 2016). Most of the record-breaking applications of these neural networks are part of the machine vision area. But they have also found applications in biological sequence analysis and speech recognition.

Most of the actual applications in semantic segmentation, object recognition, image classification, and image retrieval use CNNs. In biomedical image segmentation, they have found applications at the pixel level in embryo image processing, detection of mitosis in breast histology images, neuronal structure segmentation in electron microscopy images, and tissue segmentation in MRI scans. Segmentation applications at the image level include classification of colon histopathology images into cancerous and noncancerous, segmenting and classifying yeast microscopy images, counting bacterial colonies on agar plates, and brain structure segmentation in MRI scans (Dolz et al., 2016; Angermueller et al., 2016).

Current advances in parallel computing, optimization techniques, and network architectures have enabled recurrent neural network (RNN) applications in large-scale deep learning problems (Lipton et al., 2015). They are inspired by the cyclical connection of the neurons inside our brain. Neural networks with cyclical connections also include recursive and feedback networks (Graves, 2012). Novel applications for recurrent networks include unsupervised video encoding, video captioning, biological sequence analysis, and program execution. Moreover, most of the advances in recurrent neural networks come from new architectures rather than new algorithms.

2.5.2 Frameworks for implementation

There exist various frameworks available to implement neural networks. Some of those tools include TensorFlow, Theano, Torch, Caffe, Neon, Deeplearning4j[5] (DL4j), and the Computational Network Toolkit Abadi et al. (2016).

[5] https://deeplearning4j.org/index.html

Framework	**Core**	**Features**
DL4j	Java	Import neural networks from other frameworks
OpenCV	C++	Complete library with computer vision capabilities
TensorFlow	C++	Heterogeneous distributed computing, flexible multi-GPU support
Theano	Python	Custom models, RNNs, Ease of use with wrappers
Torch	Lua	Functional extensionality, RNNs, Reuse of existing models
Caffe	C++	CNNs, Computer vision, Support for pre-trained models
Neon	Python	Good speed, No multi-threaded CPU support

Table 2.1: Open source frameworks available for deep learning implementations. (Min et al., 2016; Angermueller et al., 2016; Bahrampour et al., 2015)

The image processing library OpenCV Bradski and Kaehler (2008) also has a machine learning library as one of its five main components (See Table 2.1).

Given the amount of data processed by modern deep learning applications, it is important to have distributed data processing platforms for deep neural network implementations. Functional programming provides inspiration for dataflow distributed platforms, where state transitions represent the processing of data. Standard dataflow systems – such as Spark and MapReduce – use directed acyclic graphs to model computations. However, those standard systems fail to scale because mutable states and iterations are crucial for deep neural network tasks, especially training operations. Because of this, parameter server systems – including DistBeleif and PMLS (Parallel ML System) – were proposed for deep neural network models (Zhang et al., 2017).

Advanced dataflow platforms – such as TensorFlow and MXNet – describe data processing as cyclic graphs with mutable states. They can also mimic parameter-server systems functionality. Once graphs are specified, advanced dataflow platforms translate them to executable models for subsequent execution. Those cyclic graphs represent symbolic computational models that can be partitioned, rewritten, and placed on distributed nodes for optimized performance. The symbolic nature of the graphs makes them slower than other dataflow systems like Spark or PMLS. This happens because of the high level of abstraction involved in the symbolic graph models. However, they are more flexible, allowing execution of deep learning models in multiple platforms. Also, they enable executions on different types of processors, including CPUs, Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs) (Jouppi et al., 2017; Zhang et al., 2017).

2.5.3 Bacteria related implementations

Bosco and Di Gangi (2016) compared two different types of deep learning models for automatic classification of bacteria species. Classification is performed without a previous feature extraction process. They propose convolutional neural networks and recurrent neural networks. For recurrent neural networks, authors use the long-short-term memory (LTSM) model. For testing purposes, the deep learning architecture was fed with a dataset encoded according to the IUPAC nucleotide representation. The dataset is downloaded from the Ribosomal Database Project (RDP) (Cole et al., 2013).

The dataset uses the 16S ribosomal RNA to identify five ordered taxonomic ranks (Phylum, Class, Order, Family, and Genus). One architecture implements a separate neural network for each taxonomic rank, while the other architecture uses multitask learning by adding separate layers for each taxonomic rank on the top of the processing layers. Their research allowed to conclude that, overall, LSTMs have a better classification output than CNNs. Another conclusion states that multitask learning improves the performance of LSTM architectures, but it harms CNN models. They way the system transforms the one-hot vector representation into a fixed length vector neglects important information in the nucleotide sequence. Moreover, the ability of RNNs to handle sequences of varying sizes is not exploited because a max pooling operation is conducted on the input sequence. Because of this, each sequence in the dataset ends up with the exact same length (Bosco and Di Gangi, 2016).

Rizzo et al. (2015) proposed a machine learning architecture for DNA sequence classification. Its implementation uses Theano, a python library for artificial intelligence models. The system proposed is a convolutional neural network for processing RNA sequences. The representation of RNA sequences is performed using k-mers occurrences. The goal is to take advantage of convolutional neural networks ability to extract hidden features. Thus, the model can extract features represented by k-mers from input sequences.

As far as testing is concerned, Rizzo et al. (2015) chose 16S rRNA sequences. They compare the proposed architecture with four existing machine learning models, including Naive Bayes (NB), Random Forest (RF), Support Vector Machine (SVM), and a previous model developed by the team. The previous model by the same team is a classifier based on a General Regression

Rank	**Accuracy RNN**	**Accuracy CNN**
Phylum	0,992 ± 0.007	0.981 ± 0.007
Class	0,990 ± 0.008	0,978 ± 0.008
Order	0,941 ± 0.029	0,908 ± 0.021
Family	0,897 ± 0.023	0,851 ± 0.024
Genus	0,733 ± 0.030	0,692 ± 0.024

Table 2.2: Bosco and Di Gangi (2016) bacteria classification system results for 16S rRNA data, considering means and standard deviations for the multitask learning tests.

Neural Network (GRNN) model. The research concludes that CNN models get very good results at the different taxonomic levels. CNNs outperformed NB, RF, and SVM classifiers when processing full-length 16S rRNA sequences or 500bp fragments. Nonetheless, results obtained by the CNN are very similar to those obtained by the GRNN model. This happens when the models are processing full-length 16S rRNA sequences. When the models process 500bp 16S rRNA sequence fragments, the CNN architecture manages to outperform all the other models. That is quite important because in metagenomics, researchers usually have access only to fragments of DNA sequences.

2.5.4 Other biological sequence implementations

Proteins are essential for life. They perform cellular tasks and give cells their structure. Twenty amino acids represent the building blocks that cells use to construct proteins. Hence, predicting protein function from amino acid sequences is an active area of research. Proteins that share similar functions are grouped into families (Lee and Nguyen, 2016; Lodish et al., 2004). The protein family classification system in (Lee and Nguyen, 2016) retrieves protein information from the Universal Protein Resource (UniProt) database. Global Vectors for Word Representation (GloVe) is a distributed representation that provides the encoding for amino acid sequences. The reference classifier is a support vector machine (SVM) with a radial basis kernel. An RNN architecture forms the base for the protein family classification system. Similar to the approach in (Park et al., 2017), the classification model includes intermediate results from the RNN into the final classification layer. The RNN with 100 hidden units outperforms both the reference classifier and a CNN-based architecture.

The bidirectional architecture suggested by (Bosco and Di Gangi, 2016) can be found in (Liu, 2017) implementation. His research work considers a bidirectional recurrent neural network to take advantage of context information. The neural network represents a sequence processing application that predicts four protein functions: iron sequestering proteins, cytochrome P450 proteins, serine and cysteine proteases, and G-protein coupled receptors. The system does not perform any feature extraction on the amino acid sequences. Instead, it downloads raw sequences from UniProt and then, the prediction system feeds them to a bidirectional LSTM. SwissProt or Uniprot annotated data – or laboratory experiments when no annotation is available – are used for validation purposes. There is only one bidirectional layer to take advantage of context information in the input amino acid sequences, which are divided 80% for training and 20% for testing. Liu (2017) represents each sequence using one-hot encoding and pads every sequence to a fixed length. Therefore, the bidirectional RNN does not process variable input sequences. Also, sequences are 333, 500, or 800 residues long, which is similar to the sequence lengths found in (Bosco and Di Gangi, 2016) when processing the 16S rRNA dataset used for bacteria classification.

Plasmids are circular, double-stranded DNA sequences. They are not part of the chromosomal DNA. However, they are replicated in the reproduction of the cell. Plasmids have a parasitic or symbiotic relationship with bacterial cells and some lower eukaryotic cells (Lodish et al., 2004). These circular DNA sequences constitute mobile genetic elements that have an active role in bacteria adaptation to environmental conditions. PlasFlow (Krawczyk et al., 2018) is a system to identify if sequences from metagenomic samples pertain to plasmids or chromosomal DNA. Regarding sequence representation, PlasFlow encodes the genomic information using k-mers of 3-7 bases. Sequences with less than 1000 bases were discarded. The neural network has an MLP architecture with one or two hidden layers. To train the MLP, data from a number of sources, including the NCBI database, was split into training and testing sets. The classification system is implemented in TensorFlow. Also, an AdaGrad optimizer minimized the loss function. PlasFlow yields more accuracy and a lower false positive rate than existing software tools like cBar (Krawczyk et al., 2018).

Micro RNAs (miRNAs) are one of the post-transcriptional control mechanisms that cells use to control gene expression. They regulate the translation of specific target messenger RNAs (mRNAs) (Lodish et al., 2004). Park et al. (2017) use one hot encoding to compute the representation for precursor micro RNAs. The encoding takes into account both the nucleotide information and the secondary structure for the sequence. Two LSTM layers and a dense module comprise the classification model. Three fully connected layers constitute the dense module. The dense module processes the output from the LSTM layers and feeds the binary classification layer. For further performance, the classification model adds an attention mechanism, which complements the last LSTM output with intermediate LSTM results. Because of the high imbalance in the dataset, balanced class weights are the mechanism that helps to calculate the loss while training the system. Results show that the neural network architecture has better prediction results than other alternatives. Also, it does not require hand-crafted feature extraction.

DeepTarget (Lee et al., 2016) represents a novel approach for miRNAs target prediction. It does not use any of the more than 151 features of miRNA available in the literature. Yet, the system manages to get a 25% increase in performance when compared to existing miRNA processing architectures. miRNAs represent short sequences of ribonucleic acids that control the expression of target messenger RNAs (mRNAs). The proposed architecture is trained to reject false positives in miRNA-mRNA pairs prediction. It does not depend on sequence alignment operations, which make it less susceptible to changes in parameters.

Cheng et al. (2016) proposed a new system for miRNAs target prediction: miRTDL. It uses convolutional neural networks to explore miRNA regulatory mechanism and identify its real targets. This analysis is very important because miRNAs regulate genes that are associated with different diseases. The convolutional network at the core of the proposed architecture has six layers. One input layer, two convolutional layers, two subsampling layers, and an output layer. It also includes a loss function to properly analyze the significance of each type of feature. Because of this, the system can better understand miRNA-mRNA interactions. To assess the efficiency of the system, research included comparisons to other machine learning methods. Cheng et al. (2016) claimed that they outperformed existing target prediction algorithms.

Giang Nguyen et al. (2016) proposed a deep learning model based on convolutional neural networks to perform DNA sequence classification. DNA sequences are modeled as text and minimum preprocessing is needed to feed the neural networks. Instead of using a feature extraction process, they encoded the DNA sequences using the one-hot vector representation. Such encoding converts genomic sequences into a two-dimensional numerical matrix. Because of this, authors preserved essential information of nucleotide position in the sequence. The numerical matrix then feeds the convolutional neural network for conducting the classification. The proposed model got performance improvements in all twelve DNA sequence validation datasets used for testing and comparison.

To deal with the black box nature of deep learning models, Lanchantin et al. (2016) created DeMo dashboard, a Deep Motif dashboard aim at visualizing motifs – genomic sequence patterns – from deep neural network models. The proposed system supports three different kinds of neural network architectures: convolutional, recurrent, and convolutional-recurrent networks. Deep learning models under consideration are designed to classify transcription factor binding site (TBFS). Authors found that convolutional-recurrent neural networks have the best performance in TBFS classification. Such performance boost happens because the hybrid architecture models both motifs and dependencies between them.

Cancer is one of the most deadly diseases affecting humanity. It represents 14.6% of deaths each year. Therefore, Yuan et al. (2016) created DeepGene, a deep learning based classifier for somatic point mutation cancer classification (SMCC). DeepGene has two preprocessing stages and one deep learning stage. As DNA sequences have large amounts of genes, the first stage filters input data and chooses a small discriminatory subset. Yet, the resulting subset is quite sparse. Thus, the second stage processes the discriminatory subset and generates a list of indexes containing only the informative point mutations. The list of indexes then feeds the deep neural network in the third stage for cancer classification. DeepGene has an increase of at least 24% in accuracy when compared to other SMCC systems.

Alipanahi et al. (2015) designed DeepBind, a deep convolutional neural network-based system to predict binding affinity of a protein to a DNA or RNA sequence. This system uses two stages: a convolutional neural network for learning representation and a prediction stage for high-level feature inference. DeepBind was tested with almost a thousand publicly available datasets. Results obtained by DeepBind indicate a nearly perfect accuracy. The model manages to consistently outperform existing methods, even though some of them are based on extensive biological knowledge. Such accuracy proves the architecture ability to learn higher level features because the authors only give low-level features explicitly.

Chapter 3
Sequential Deep Learning System

Adaptive systems create, update, and use internal models of their environment to make predictions; successful models create valid homomorphisms of their environment

(Forrest and Mitchell, 2016)

The classification of bacteria using their whole genome sequences is a part of a broader set of tasks in machine learning: **sequence labeling** (Graves, 2012). The remaining of this section is based on Graves (2012).

In sequence labeling, an input sequence of data points is transcribed with a discrete set of output labels. Examples of real applications include speech recognition, gesture recognition, and protein secondary structure prediction. It is common to have sequence labeling tasks in time series data, where each data point in the input sequence is a time step. Nonetheless, the same approach works for non-temporal tasks such as protein secondary structure prediction and DNA sequence classification. When considering biological sequences, each time step represents a residue or a nucleotide.

In sequence labeling applications, the input sequence vector is considered to have a higher dimension than the output label vector. For some problems, it is important to infer the exact time step of the label. Thus, the learning algorithm should discover the exact alignment between input sequences and output labels. However, the classification of DNA sequences only takes into account the output set of labels.

If the input sequences are independently and identically distributed, we have a classical pattern classification problem where sequences are the patterns in the model. Although you can make the same assumption in problems when sequences do not have such distributions, as long as you set appropriate boundaries to each sequence. In any case, data points inside each sequence are never assumed to be independent.

Considering applications were the exact alignment between input sequences and discrete labels should not be learned by the model, we have three different types of sequence labeling problems: sequence classification, segment classification, and temporal classification.

Sequence classification is the most restrictive case of sequence labeling applications. The input sequence should be matched to a label sequence of unitary length. But this case has an advantage over the other cases because the algorithm can process the whole sequence before generating the label. This kind of classification is what we perform with the bacterial whole genome sequences. Other examples include a word in speech recognition or a person who wrote a letter. If the input sequence can be padded to a predefined length, the problem becomes a classical pattern classification application, and any type of discriminant or probabilistic method can be applied to the input data set. Even in this case, sequential processing by RNNs can offer the advantage of a better response to distortions and shiftings in input sequences. Sequential models not only have the advantage of robustness against distortions and shifting in the input sequence, but also have the ability to discover what context information is essential for a correct output, as they can acquire a better knowledge of the overall sequence structure.

The classical error rate in pattern classification – the classification error rate – enables the computation of a loss function for the models. The error rate is the relationship between correct input-target outputs and the testing set size:

$$E = \frac{1}{|S'|} \sum_{(x,z)\in S'} \begin{cases} 0 \text{ if } h(x) = z \\ 1 \text{ otherwise} \end{cases} \tag{3.1}$$

where S' is the test set, x is the input, z the expected output, and h is the learning algorithm.

Segment labeling is the set of problems where the algorithm should generate a label for a set of data points in the input sequence. The alignment between the input sequence and target labels needs to be exact. These type of applications appear in natural language processing and bioinformatics, where input sequences are discrete, thus, they are easily segmented. Other types include audio and image processing. In this case, the segmentation is not trivial and a manually segmented training set should feed the learning algorithm to get correct results. Segment error rate measures the relationship between the hamming distance of generated and expected label sequences and the total count of training label sequences

Temporal labeling is the most general case where the label sequence can vary in length - it could have a zero length - and its alignment to the input sequence is not important. The only restriction is that the label sequence length should be less than or equal to the input sequence length. For this kind of problems, hybrids of hidden Markov models - RNNs are usually a good alternative. There also exists an approach that uses only RNNs. The label error rate provides error measurements for this type of learning algorithms. It is the relationship between edit distance of output and expected label sequences and the total count of training samples. Edit distance refers to the number of insertions, deletions, and substitutions necessary to transform one sequence into the other. If the label sequence in a temporal classification task should be exact, the label error rate does not work. Instead, the sequence error rate should be used for such applications.

3.1 Recurrent Neural Networks

Sequences found in biology naturally fit the processing power of recurrent neural networks (RNNs). That happens because of the temporal modeling capabilities of RNNs. RNNs are inspired by the cyclical connection of the neurons inside our brain. They store information from input sequences by using iterative function loops. RNNs are an ideal architecture for sequence labelling tasks because they store context information in a flexible manner. By learning what to store and what to ignore, they accept input data in different types and representations. Also, They can understand sequential patterns in the presence of sequential noise (Graves, 2012).

A time window approach used by other nonsequential networks suffers from lack of robustness against sequential distortions and the need to manually determine the window length. It also increases the number of weights in the network. Another alternative is to introduce a delay from input processing to output generation. This alternative is robust against sequential distortions but the delay sequence should be manually determined. Also, the network should remember original inputs throughout the delay (Graves, 2012).

A useful approach to better understand RNN architectures is to unfold the cyclical connections into a graph, where each time step forms a node and shares the same weights as other nodes (Graves, 2012). Indeed, RNNs are powerful architectures. Hyötyniemi (1996) claims that any computable function can be implemented by a recurrent neural network. As Turing machines can also implement this type of functions, the author proves their equivalence by comparing an RNN based on perceptrons with a program implementing a computable function. The equivalence is found in terms of the network internal state to the program state and the transitions of states to the program flow.

Recurrences (Lehman et al., 2010)

In math, recurrences are sequences of numbers where the first terms are defined as constants, while the subsequent terms are expressed as a function of the first ones. For instance, the Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21 ... can be described by the following recurrence:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \text{ for } n \geq 2 \end{aligned} \tag{3.2}$$

Recurrences are examples of a common pattern in computer science: the divide and conquer approach. This approach helps to solve problems by decomposing them into smaller ones until reaching easy base cases. Examples include the analysis of recursive algorithms, enumeration of structures, and analysis of random processes.

There exist a number of architectures used by RNNs. When the alignment between input data and discrete labels is necessary, Graves (2012) proposed the connectionist temporal classification (CTC) to label input sequences with unknown alignments. For input data that has more than one dimension, such as 2D images, 3D images, videos, and Magnetic Resonance Image (MRI) scans, conventional RNNs are extended to include more dimensions. The extended networks receive the name of Multidimensional RNNs (MD-RNNs). They have been used in applications including image captioning, video captioning, and image segmentation (Stollenga et al., 2015).

In some applications regarding sequence analysis, the current output depends on both past inputs and future inputs. RNNs process input data in one direction, thus, context information is available from one side of the sequence. For instance, in time series data, context is available from the past but there is no way data after the current point in time can alter the context. Therefore, bidirectional RNNs (BRNNs) were created to cope with those sequence analysis requirements. In this kind of networks, we assemble two hidden layers, one processes the sequence from the first input while the other processes the sequence from the last input. The output layer is connected to both backward and forward hidden layers and it only generates the output after processing the whole sequence (Lipton et al., 2015; Graves, 2012).

The BRNN is an elegant solution when compared to other approaches. It has successful applications in protein secondary structure prediction and speech recognition, outperforming standard RNNs. For causal tasks, we could think that BRNNs violate the principles of causality. This is true in some temporal applications such as robotics or financial time series, where there is no sense in expecting future inputs for a model. Spatial sequences do not have such restriction because usually, we have the whole input sequence before starting the processing. For instance, in biological sequence processing or image segmentation, you have all the data beforehand. Nonetheless, some temporal sequence processing tasks could also take advantage of the BRNNs robustness. Those temporal applications include tasks where the output is expected at the end of some input segments, like words in natural language processing or speech recognition. Also, realtime temporal applications could also be considered as long as they have natural breaks in the input sequence and the output can wait for such breaks to process the input segment. (Graves, 2012).

Long Short-Term Memory (LSTM) systems were proposed back in 1997 by Hochreiter et al. (Hochreiter and Schmidhuber, 1997). It is a useful approach for information storage in machine learning models. The system is based on gradients to efficiently learn in recurrent backpropagation neural networks. Its computational complexity in big-O notation is constant. LTSM architectures achieve a faster learning rate than existing models such as Elman nets. LSTMs are a redesign of RNNs around a memory cell. The redesign improves their ability to store context information on very long sequences. In fact, they solve complex long-time-lag tasks that have never been solved by existing models (Graves, 2012).

Gated Recurrent Units (GRUs) represent another type of gated RNNs (Goodfellow et al., 2016). They are motivated by LSTMs but are simpler to compute and implement. Also, they can dynamically control the time scale and forgetting behavior of the units in the network. Units comprise update and reset gates. Update gates select if the hidden state will be updated with a new hidden state. On the other hand, reset gates determine if the previous hidden state will be ignored (Cho et al., 2014b; Goodfellow et al., 2016). Both LSTM and GRU networks share the ability to better model and learn long-term dependencies. Therefore, GRUs could replace LSTMs without affecting network performance (Bahdanau et al., 2015).

Even though the addition of memory cells in the LSTM neural networks give them the ability to process very long sequences of input data, they usually struggle when dealing with such long sequences. The processing time usually increases as well as the memory consumption. Therefore, Graves (2012) proposed an improved architecture for such very long sequences of input data: Hierarchical subsampling RNNs. These hierarchical neural networks have a stack of recurrent layers, each one with a smaller size than the previous layer. The increasingly smaller size helps to diminish the amount of memory consumption and improve the neural network processing time.

Another variation of RNNs is the Neural Turing Machine (NTM) (Graves et al., 2014). It uses concepts from biological memory inner workings and known digital computer architectures. The NTM is an extension of the RNN that includes addressable external memory (Lipton et al., 2015). The proposed model is able to learn algorithms from example data. Then, the model can generalize algorithm application when exposed to natural data. Gradient descent can properly train the proposed architecture. To test the proposed model, experiments for copying and sorting data sequences were implemented.

3.1.1 Unit equations

Given an input sequence $x_1, x_2, \dots x_T$, equations for a standard RNN are defined as follows (Martens and Sutskever, 2011):

$$\begin{aligned} t_i &= W_{hx}x_i + W_{hh}h_{i-1} + b_h \\ h_i &= e(t_i) \\ s_i &= W_{yh}h_i + b_y \\ y_i &= g(s_i) \end{aligned} \tag{3.3}$$

where W_{hx}, W_{hh}, W_{yh} are learnable weight matrices, b_h, b_y are biases, h_i are the hidden states, y_i are the outputs, e and g represent activation functions. Common activation functions for RNNs are the hyperbolic tangent (Equation 3.4) and the logistic sigmoid (Equation 3.5). Both functions are vector valued functions which are differentiable and non-linear. A differentiable activation function allows the use of gradient descent for neural network training. On the other hand, non-linearity makes neural networks more powerful than their linear equivalents (Graves, 2012; Martens and Sutskever, 2011).

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.4}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

For the network outputs in classification systems with more than two classes, a softmax function (Equation 3.6) provides normalized output activations that represent the class probabilities (Graves, 2012).

$$y_j = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_j}} \text{ where } j = 1, 2, \dots, K \tag{3.6}$$

True class probabilities are obtained by representing the true labels with a 1-of-K coding scheme (Cho et al., 2014a), a binary vector with one-hot encoding (Section 3.2). The cross entropy loss function (Equation 3.7) provides the target function that we minimize in order to train the network. By doing so, we minimize the classification error rate in Equation 3.1.

$$\mathcal{L} = -\sum_{k=1}^{K} z_k ln(y_k) \tag{3.7}$$

where z represents the true class probabilities and y represents the network output probabilities.

The following equations represent an LSTM unit (Graves, 2013):

$$\begin{aligned} i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\ c_t &= f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\ h_t &= o_t tanh(c_t) \end{aligned} \tag{3.8}$$

where i is the input gate, f is the forget gate, o is the output gate, and c is the unit cell. b are biases and W are learnable weight matrices. The hidden state of the LSTM is the concatenation of h and c.

GRUs are similar to LSTM units. However, they are simpler and can outperform LSTMs on various tasks. GRU equations are defined as follows (Jozefowicz et al., 2015; Cho et al., 2014a):

$$\begin{aligned} r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \tilde{h}_t &= tanh(W_{xh}x_t + W_{hh}(r_t h_{t-1}) + b_h) \\ h_t &= z_t h_{t-1} + (1 - z_t)\tilde{h}_t \end{aligned} \tag{3.9}$$

where r_t is the reset gate, z_t is the update gate, b are biases and W are learnable weight matrices.

3.2 Sequence representations

A one-hot vector (Giang Nguyen et al., 2016) representation of DNA or protein sequences is used as the input to machine learning systems. The one-hot vector encoding converts genomic or protein sequences into a two-dimensional numerical matrix. Because of this, the encoding preserves essential information of nucleotide or residue positions in the biological sequence. But it is important to have in mind the sparsity that one-hot vector encoding adds to the sequence representation. Depending on the length of the sequence, this representation can also suffer from high dimensionality (Ng, 2017).

Another digital encoding, the k-mers representation (Rizzo et al., 2015), allows researchers to achieve a fixed length representation of biological sequences using occurrences of overlapping subsequences with a length of k. The mapping achieved by k-mers is similar to the feature extraction that happens in image processing. K-mers are small DNA or protein sequences of k length. Based on the occurrence of those small sequences, the system computes and spectral representation of input data. The spectral representation then feeds the neural network architecture for a number of purposes. In fact, k-mers represents a powerful approach and they are currently used in multiple metagenomics and taxonomic methodologies. They are an excellent alternative to represent input sequences for the subsequent processing by deep neural networks.

Hasman et al. (2013) employ k-mers to perform bacteria identification. Sequences come from isolated bacterial samples found in urine samples. The identification method uses k-mers, more specifically 16-mers, which are extracted from 1647 complete bacterial genomes. All the genomes come from the NCBI database. A database is created to store the 16-mers information. In order to decrease the database size, only 16-mers starting with ATGA are considered. By doing so, the 16-mers database has a roughly 256-fold reduction in size. This is important because 16-mers account for 4^{16} possible words. Once the input file of the target bacterial sequence is generated, the method extracts unique 16-mers from the file. Then, it scans the database and computes the number of exact matches of 16-mers per entries. The result depends on the species that yields the highest amount of hits in the database search.

An important aspect of sequence labeling problems is the use of context in the input sequence. Context is fundamental to generate the correct label sequence. For non-sequential learning algorithms, which processes one input at a time, the input sequence could be sampled in time windows, an each time window becomes an input to the model. But this approach has a serious flaw: we can not know beforehand what the extent of the important context information is, and the extension can vary between different input sequences. Thus, it is impossible to determine beforehand the appropriate length of the time window (Graves, 2012).

Based on results from Natural Language Processing (NLP), where the use of context information improves the performance of neural network applications, the seq2vec (Kimothi et al., 2016) extends the use of a single k length in a k-mers representation, to create a distributed representation of the sequences in a Euclidean space. The distributed representation, which considers different k's, has the potential to capture contextual information in the original sequence. The dna2vec system (Ng, 2017) uses a distributed representation as well, considering $k = \{3, 4, 5, 6, 7, 8\}$ to generate vectors in a 100-dimensional space. The cosine similarity of those vectors is correlated to the Needleman-Wunsch similarity score.

Even though the distributed representation does not have the positional information that provides the one-hot vector encoding, it provides contextual information and helps to deal with high dimensionality in the sequences – an important aspect considering that neural networks benefit from reasonably compact continuous vector inputs (Ng, 2017; Graves, 2012). An additional benefit is the invariance of the distributed representation to the order of the nodes in the FASTA file. Although scaffolds are ordered in the FASTA file, contigs are not necessarily ordered.

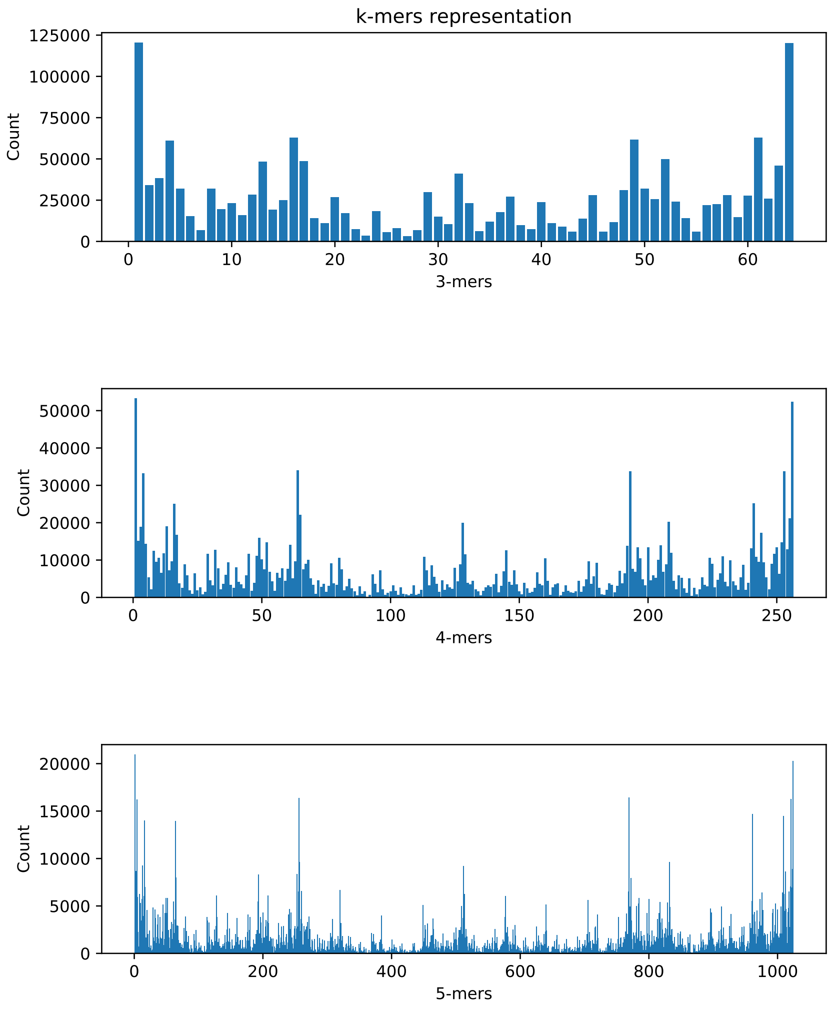


Fig. 3.1: k-mers example representation for a sequence pertaining to species *Campylobacter Jejuni*. The representation considers histograms for $k = \{3, 4, 5\}$

Thus, we use a distributed representation for encoding the bacterial whole genome sequences. Our representation takes the idea of multiple k-mers from dna2seq, but we consider a group of three k's, namely $k = \{3, 4, 5\}$ (Fig-

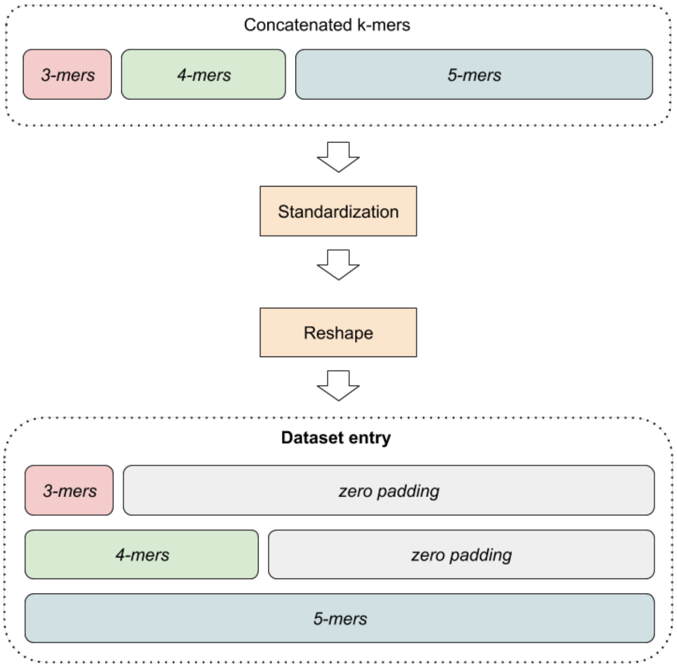


Fig. 3.2: Distributed representation based on k-mers, with $k = \{3, 4, 5\}$. First, histograms are concatenated into a 1x1344 vector to perform standardization. Mean subtraction and division by sttandard deviation comprise the standardization process. Then, k-mers histograms are zero padded and stacked to form a 3x1024 matrix representation.

ure 3.1). Using the original set of k's in dna2seq would have generated a concatenated vector with 87360 dimensions. But our set of k's generates a concatenated vector of only 1344 positions. Once the concatenated vector of histograms is calculated, we proceed to the standardization of the data, which is also known as scaling (Graves, 2012; Angermueller et al., 2016). The standardization sets the mean to zero and the standard deviation to one for every dimension in the input vectors. Means and standard deviations are calculated on the training set. Then, those values are used for standardizing training, validation, and testing vectors. After standardizing input vectors, histograms for each k-mer are padded with zero to the length of the largest histogram and stacked into a matrix of 3x1024 (Figure 3.2).

3.3 Neural network training and evaluation

The deep learning model training takes into account the following aspects: optimization methods to minimize the error, parameter initialization and optimization, learning rate and batch size, learning rate decay, momentum, per-parameter adaptive learning rate methods, batch normalization, analysis of the learning curve, and monitoring training and validation performance (Angermueller et al., 2016). Table 3.1 presents the hyperparameters that are usually considered for deep learning architectures. It also contains default values for each hyperparameter that we will take into account for training our model.

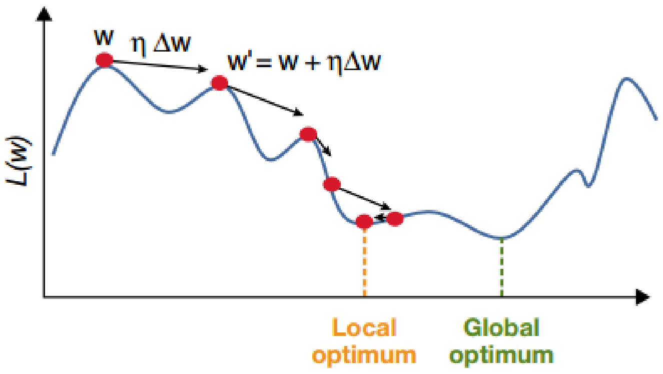


Fig. 3.3: Stochastic Gradient Descent with varying learning rate η. From (Angermueller et al., 2016)

$$\frac{\partial f(x,y)}{\partial x} = \frac{\partial f(x,y)}{\partial y}\frac{\partial y}{\partial x} \tag{3.10}$$

Stochastic gradient descent (SGD) is a very successful method for training deep neural models (Figure 3.3). It is a popular approach in the optimization field. The gradient provides the direction of maximum change of the loss function. As we are minimizing the loss, we move in the opposite direction of the gradient. Small variations of the gradient are added to the function parameters every iteration. By doing so, the function moves towards the global minimum – ideally. We compute the outputs of the neural network from the training set. However, the loss is expressed in terms of the neural network weights. To compute the gradient of the loss function, the chain rule (Equation 3.10) provides an effective tool to perform the computation by using the calculated outputs. This procedure is also known as a backpropagation.

Parameter	**Range**	**Default value**
Learning rate	0.1, 0.01, 0.001, 0.0001	0.01
Batch size	64, 128, 256	128
Momentum rate	0.8, 0.9, 0.95	0.9
Weight initialization	Normal, Uniform, Glorot uniform	Glorot uniform
Per-parameter adaptive learning rate methods	RMSprop, Adagrad, Adadelta, Adam	Adam
Batch normalization	Yes, no	Yes
Learning rate decay	None, linear, exponential	Linear (rate 0.5)
Activation function	Sigmoid, Tanh, ReLU, Softmax	ReLU
Dropout rate	0.1, 0.25, 0.5, 0.75	0.5
L1, L2 regularization	0, 0.01, 0.001	–

Table 3.1: Hyperparameters to have into account for training deep learning architectures. Both ranges and default values are included for each hyperparameter. From (Angermueller et al., 2016)

Backpropagation through time (BPTT) enables the use of backpropagation in sequential architectures. Once we have determined the partial derivatives of the loss function with respect to the neural network outputs, BPTT allows the computation of partial derivatives with respect to the neural network weights. Moreover, BPTT represents a good alternative from the standpoint of computational complexity and execution time, although it consumes more memory than real-time recurrent learning (RTRL), another option for determining the neural network weights (Graves, 2012).

Variations of the SGD method include the addition of momentum to increase the speed of convergence, and the use of a constant to multiply the gradient every iteration. Such constant is known as the learning rate. Alternatives to SGD include RMSProp, AdaGrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2014) optimization methods. Although all of the three alternative methods use an adaptive learning rate, Adam combines the advantages of RMSProp and AdaGrad optimizers. Adam stands for adaptive moment estimation because it relies on the first and second moments of the gradient. Based on those moments, this optimization method computes adaptive learning rates for every trainable parameter. It is usually the recommended optimizer for many applications (Angermueller et al., 2016).

3.3.1 Regularization

It is important to avoid model overfitting to training data in order to allow the model to generalize its classification capabilities to future genomic sequence inputs. The calculation of parameters to minimize the loss function uses the input training set. But we need a good inference performance when the network process data that it has not seen before. The extrapolation of performance from the training set is known as generalization and it's a crucial factor in the machine learning field. There exist a number of methods to improve generalization of neural networks when training them with a fixed size input set. Such methods are known as regularizers. Simple regularizers include early stopping, weight noise, and input noise (Graves, 2012).

Early stopping occurs when evaluating the stop conditions for the training steps or choosing the best weights for the neural network after steps are completed. This approach uses a validation set alongside training and testing sets. The validation set is usually 10% of the data available. When training the network, the loss of training, testing, and validation sets fall sharply at the first iterations. After that decrease in loss values, validation and test set losses stabilize or start to slowly increase, while the loss on the training set continues to decrease. This phenomenon is a clear sign of overfitting. Thus, the best weights minimize the loss function on the validation set before it starts to slowly increase. A disadvantage of this approach is the decrease in the size of the available samples in the training set (Angermueller et al., 2016; Graves, 2012).

Weight noise consists in altering the weights with Gaussian noise. A zero mean is used and the noise addition happens in each sample from the training set. This method does not depend on the input data. But it is less effective than input noise and it can also affect the convergence of the training process. Weight noise helps to simplify the network, and the simpler the network, the better. Neither weight noise nor input noise should be added when evaluating the network on the test set (Graves, 2012).

The size of the input dataset is important because neural networks are data intensive architectures. Input noise consists in adding Gaussian noise to inputs in order to artificially augment the number of training samples. It also improves the robustness of the neural networks. The noise should be generated for every training sample and no noise should be reused. The variance when adding Gaussian noise is difficult to determine beforehand – the mean should be zero all the time. A plausible approach is to use the validation set for estimating it. Also, the noise should ideally model variations observed in real data. Unfortunately, such models usually do not exist (Graves, 2012).

Data augmentation is a common approach in image processing. Gaussian noise would create a speckled version of the images. Instead, images are rotated, scaled, or translated to increase the size of the dataset. For biological sequence applications, rotations, scaling, or translations do not have much sense because we are dealing with 1-dimensional sequences. We have to consider perturbations that have sense from a biological point of view. Models already exist for scoring variations in both amino acid and nucleotide sequences. Such models receive the name of similarity-scoring matrices (Pearson, 2013; Graves, 2012). They give us a clue of variations that we can perform on training data. Therefore, a plausible alternative is the artificial mutation of individual nucleotides using scoring matrices as a guide. The number of nucleotides that can mutate per sequence is critical because we can not significantly alter the identity percentage. We should respect identity percentage thresholds among sequences of the same species.

Another regularizer is the dropout technique (Srivastava et al., 2014). It is a methodology to improve the training process of neural networks by reducing overfitting. The purpose of the method is to bypass the adaptations of the deep learning model to the training dataset. Standard backpropagation techniques generate this adaptation but the method uses random dropout to break them. In spite of the improvements achieved with Dropout, it increases the training time by two or three times. This training technique has been tested with multiple datasets. The variety of testing datasets proves the system suitability for deep neural networks in spite of the application under study. A considerable amount of applications benefited from the use of dropout in the training phase, including speech recognition, computational chemistry, and natural language processing (Dahl, 2015).

3.3.2 Metrics

Accuracy could be a poor estimator of neural network performance in some applications. Accuracy defines the relationship between the correct outputs and the total number of outputs. A more robust approach include the calculation of precision and recall. Precision defines the fraction of detections reported by the network that are correct. On the other hand, recall defines the fraction of true events in the testing set that were successfully detected by the network (Goodfellow et al., 2016; Rizzo et al., 2015):

$$Accuracy = \frac{T_P + T_N}{T_P + F_P + T_N + F_N} \tag{3.11}$$

$$Precision = \frac{T_P}{T_P + F_P} \tag{3.12}$$

$$Recall = \frac{T_P}{T_P + F_N} \tag{3.13}$$

where T_P are true positives, F_P are false positives, T_N are true negatives, and F_N are false negatives[1]. Table 3.2 has a graphical representation of T_P, F_P, T_N, and F_N for a binary classification model. Such graphical representation receives the name of confusion matrix (Tan et al., 2018). A heat map for the confusion matrix provides a graphical representation that improves the visualization of results in multi-class problems. Axes in the heat map denote actual and predicted labels.

	Predicted Class 0	**Predicted Class 1**
Actual Class 0	T_P	F_P
Actual Class 1	F_N	T_N

Table 3.2: Confusion matrix for a binary classification model

A common summarization of neural network performance combines precision p and recall r into the balanced F-score (Goodfellow et al., 2016):

$$\text{F-score} = \frac{2pr}{p + r} \tag{3.14}$$

Metrics performed over test sets allow to estimate the predictive accuracy of neural networks – their generalization performance. However, from a statistical standpoint, one metric measurement does not provide a robust characterization. Cross-validation provides a fully general and nonparametric way to assess the prediction performance of neural network models. 10-fold cross-validation is a typical configuration. It uses a training set size of 90% and a testing set size of the remaining 10% of the data. A common approach is to partition the dataset into ten groups randomly to ensure samples are drawn from an independent and identical distribution (Efron and Hastie, 2016; Tan et al., 2018). Training and testing cycles are repeated ten times, making sure neural network weights are initialized every cycle. Metrics are then expressed in terms of means and standard deviations. Also, the number of measurements for the same metric enables the use of methods for statistical significance tests.

[1] `https://www.tensorflow.org/api_docs/python/tf/metrics`

3.4 Implementation

The whole system is implemented in Python 3.6[2]. Three main modules are considered for the Python code: download module, dataset module, and classification system module.

The download module takes advantage of the *urllib.request* library to download compressed FASTA files into the local disk, using the NCBI FTP endpoint. On the other hand, the dataset module loads FASTA files from local disk and extracts a string representing sequences of nucleotides per file. The string is then processed to compute the distributed k-mers based representation (Section 3.2). It is important to note that extracting the distributed representation per sequence is time consuming. Thus, parallelization is essential for a faster processing. The *threading* higher-level Python interface enables the creation of multi-threading applications. However, the Global Interpreter Lock (GIL) – a mutex that protects access to objects – prevents the execution of multiple threads at once. Because of this limitation, it is not possible to take advantage of multi-core processor systems with *threading.Thread* class[3].

Therefore, the dataset module uses the *multiprocessing* interface. This interface is based on processes, which can be executed in parallel by multi-core processors. A subclass of the *multiprocessing.Process* class processes each string of nucleotides and generates the distributed representation. For inter-process communication, the *multiprocessing.Queue* class provides a process safe channel. Once the distributed representation is completed, the subclass sends the results back to the main process for memory storage. To avoid spanning an indefinite number of processes, a pool of processes is created and periodically executed once a threshold is surpassed.

For an even faster computation time, especially considering the number of samples that should be preprocessed, a C++ implementation performs the distributed representation computation and stores the results in a comma-separated values (CSV) file.

The C++ implementation uses constant unordered maps[4] of string-integer pairs for storing the k-mers dictionaries. k-mer words are the keys while indexes in the histograms are the values. Unordered maps in C++ provide an average constant access time, an important aspect considering the number of times that those dictionaries are accessed while generating the histograms for a nucleotide sequence. For consistency between both Python and C++ implementations, the k-mer dictionaries were printed from the Python dataset module and then used for the constant unorder map declarations in the C++ code.

[2] https://www.python.org/downloads/release/python-360/

[3] https://docs.python.org/3/

[4] http://en.cppreference.com/w/cpp/container/unordered_map

The Portable Operating System Interface (POSIX) threads are the mechanism for parallel execution in the C++ implementation. POSIX contains a set of standard abstractions to provide portability on UNIX operating systems. It has been around since the 1980s, when fragmentation in UNIX systems was a serious concern. POSIX is an IEEE standard that comprises specifications for Operating System (OS) aspects such as directory structure, command-line utilities, environment variables, and functions at the system level. It also has implementations in the C programming language for OS abstractions. Such abstractions include inter-process communication (IPC), signals, streams, threads, and sockets (Atlidakis et al., 2016).

The Pthread library in C/C++ provides support for POSIX threads in applications. Threads constitute a lightweight mechanism for parallel or concurrent execution of code. They are set of instructions that can be independently or simultaneously scheduled by the OS. They run inside a UNIX process with an independent flow of control. Although threads maintain their own copy of essential resources, they share most of the process resources while the process exists in the OS. Thus, memory access should be synchronized to avoid software malfunctions (Barney, 2017).

Results in the CSV file generated by the C++ implementation are then accessed by the dataset Python module. Once the distributed representations are loaded into memory, the Python dataset module integrates the information into the classification system. The module partitions the data into training, testing, and validation sets. Proportions for training operations, tests, and cross-validations appear in table 3.3.

Set	**Training**	**Tests**	**Cross-validations**
Training	60	80	90
Validation	20	–	–
Test	20	20	10

Table 3.3: Dataset proportion for partition into training, testing, and validation sets.

3.4.1 Deep learning module

The classification system module is implemented using the Tensorflow deep learning framework – an advanced dataflow system that provides one of the most efficient implementations for RNNs (Zhang et al., 2017; Angermueller et al., 2016). Based on their previous experience with the parameter-server system DistBelief, Google created TensorFlow, a system that expresses deep learning models as directed cyclic graphs with mutable state. Nodes in the graphs represent operations – some of them are control flow operations. On the other hand, edges in the graphs are Tensors – a grid of numbers with arbitrary dimensionality (Goodfellow et al., 2016). Other special edges without data represent control dependencies (Zhang et al., 2017).

TensorFlow is a system that includes both an interface for expressing machine learning algorithms, and a reference implementation to train and run those algorithms. While the core system was implemented in C++, client APIs are available for a number of programming languages, including Python, C++, Java, and Go. High level operations and optimizers are available in the interfaces for easily expressing complex deep neural network models. Low level operations are also available in the APIs. Google released the open source TensorFlow interface and a reference implementation to the community in November 2015. It was released as an open source project under the Apache 2.0 license (Abadi et al., 2016; Zhang et al., 2017).

The runtime of TensorFlow has three components: clients, masters, and workers. The client allows the specification of deep learning models and their execution through sessions. Once the client starts the session, it communicates to the master for graph execution. The master runs the graphs by scheduling one or more workers in the different machines available for graph execution. The workers execute the operations, which are implemented in kernels. Execution of the graph run in parallel whenever it is possible. Each round of graph execution is known as a step. For graph executions when multiple processing devices are available, the runtime has a node placement algorithm to decide which device is less costly for a specific operation. Also, there are saving operations to store Tensors in the local disk. Saving operations create checkpoints on the local disk that can be restored if the need arises. Checkpoints enable fault tolerant execution (Zhang et al., 2017).

The framework offers the possibility to execute machine learning models in distributed architectures, allowing faster training and testing of the model to be implemented. Moreover, it supports training of the recurrent neural network model using GPUs. Deep neural networks training can take hours, days, or even weeks depending on the neural architecture and the input dataset. The use of GPUs can decrease the training time by tenfold or more, a crucial factor to evaluate model variations efficiently (Angermueller et al., 2016).

NVIDIA is a popular vendor for GPUs. It also provides the Compute Unified Device Architecture (CUDA), which is a parallel computing platform and a programming model (Harris, 2017). It supports programming using C, C++, and Fortran. To execute the programs accessing the GPU, the CUDA toolkit is freely available for Windows, Linux, and Mac OSX. The CUDA toolkit[5] is a development environment to develop, optimize, and deploy applications to GPU enabled computing platforms. It also has libraries for linear algebra, image and video processing, deep learning, and graph analytics. In

[5] https://developer.nvidia.com/cuda-toolkit

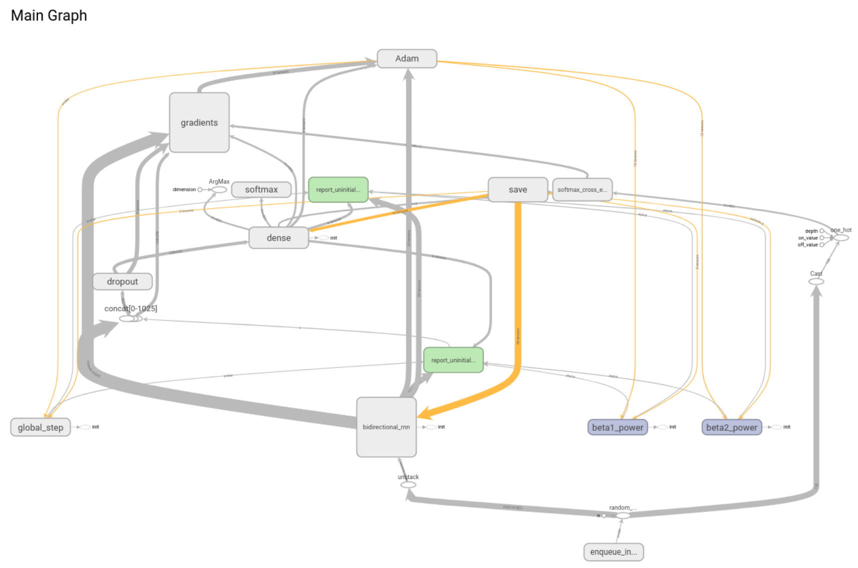


Fig. 3.4: Symbolic cyclic graph for the BRNN used in the bacteria classification system.

particular, the NVIDIA CUDA Deep Neural Network library (cuDNN)[6] allows the acceleration of deep learning frameworks and faster execution times for training the models.

Based on the recurrent neural network architecture in (Liu et al., 2016) – that processes protein sequences of up to 800 residues – our classification system has a bidirectional GRU hidden layer with 128 units. The forward and backward final states are concatenated before applying a dropout of 0.5. The model uses only a fully connected layer that takes the dropout result as input. Softmax generates the predicted labels in the fully connected layer. To initialize the weights, we use a normal distribution with zero mean: $w_{i_0} = \mathcal{N}(0, 0.1)$. Figure 3.4 depicts the symbolic cyclic graph generated for the base classification system using TensorFlow. In the graph, operations represent nodes, while tensors denote edges that show the way information flows through the system.

Results from Neural Machine Translation (NMT) prove that intermediate output states from RNN units can significantly improve the performance of the models (Luong et al., 2015; Bahdanau et al., 2015; Rocktäschel et al., 2015). Such approach has been applied in protein family classification (Lee and Nguyen, 2016) and precursor miRNA identification (Park et al., 2017). The attention mechanism (Luong et al., 2015; Bahdanau et al., 2015) creates

[6] https://developer.nvidia.com/cudnn

a context vector $\mathbf{c}_t$ from a weighted combination of intermediate output states (Equation 3.18). We use the global attention with general content-based score (Equation 3.16) in Luong et al. (2015). Weights are stored in an alignment vector $\mathbf{a}_t$:

$$\mathbf{a}_t = \frac{\exp(score(\mathbf{h}_t, \tilde{\mathbf{h}}_s))}{\sum_{s'} \exp(score(\mathbf{h}_t, \tilde{\mathbf{h}}_{s'}))} \tag{3.15}$$

$$score(\mathbf{h}_t, \tilde{\mathbf{h}}_{s'}) = \mathbf{h}_t^T \mathbf{W}_a \tilde{\mathbf{h}}_s \tag{3.16}$$

where $\tilde{\mathbf{h}}_s$ are the intermediate output states, $\mathbf{h}_t$ is the last output state and $\mathbf{W}_a$ is a learnable weight matrix. The concatenation of the context vector and the final output state into a dense layer generates the output of the attention mechanism:

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_c\,[\mathbf{c}_t, \mathbf{h}_t]) \tag{3.17}$$

$$\mathbf{c}_t = \sum_{s'} \mathbf{a}_t \tilde{\mathbf{h}}_{s'} \tag{3.18}$$

The score in Bahdanau et al. (2015) is slightly different. Instead of multiplying hidden state vectors, it adds them before computing a tanh:

$$score(\mathbf{h}_t, \tilde{\mathbf{h}}_{s'}) = \tanh(\mathbf{W}_a \mathbf{h}_t + \mathbf{W}_b \tilde{\mathbf{h}}_s) \tag{3.19}$$

where $\mathbf{W}_a$ and $\mathbf{W}_b$ are learnable weight matrices.

3.4.2 User interface

Along with the three main modules described beforehand, two additional modules provide a user interface to the classification model: a command line interface (CLI) module and a web application implementation.

The CLI interface receives the path to the file as a command argument. Then, it loads the means and the standard deviations for the subsequent standardization of the distributed representation (Section 3.2). Means and standard deviations are calculated from the training set. It also loads the mapping between labels and taxonomic IDs. This mapping is essential because the CLI module must print the species scientific name when the neural network generates its prediction.

As a first step, the CLI module ensures the file format extension matches fsa_nt.gz (the standard compressed FASTA file from GenBank). It then computes the k-mers distributed representation for the sequence. Previously loaded means and standard deviations provide the values for standardizing the distributed representation. Once the distributed representation is normalized, the module loads the recurrent neural network to predict a label for the sequence. Finally, the label is translated to the scientific name of the species. A softmax function generates a probability for the predicted label, which serves as the score of the prediction.

On the other hand, the backend of the web application serves the HTML and JavaScript files for the browser interface. It also serves a compressed version of the CLI for download. One endpoint – at */classify* – listen to POST requests from the front end side of the application. This endpoint triggers the exact same steps as the CLI interface, generating a JSON output with the scientific name of the species and the corresponding score. AJAX requests provide an easy interface to access the endpoint from the frontend.

The Flask[7] Python microframework and the React[8] JavaScript library are the main components supporting the web application. On top of that, Bootstrap, a component library based on HTML, CSS, and JavaScript, provides visual components for the frontend. Bootstrap also accounts for the flexibility of the web application for adapting to different screen sizes – making it accessible and responsive in mobile and desktop devices.

Four views comprise the web application: a home view that welcomes the user and provides a brief description of the classification system. A classify view with the form to upload the FASTA files plus a component to display results. A download view with a link to the compressed CLI file and the installation instructions. And a species view containing a table with all the scientific names of the species that the classification system supports, plus taxonomic IDs.

All the interfaces and the model files are stored in a Docker[9] container for easy installation on servers or virtual machine instances. The container is publicly available at Docker Hub[10].

Docker containers provide an efficient packaging and deployment solution. They enable the packaging of the application and its dependencies into a self-contained image. For consistency, Dockerfiles offer a declarative definition to standardize the building of the image, a paradigm known as IaC (Infrastructure-as-Code). In contrast to virtual machines, that virtualizes the server and its OS, containers are lightweight packages that run on top of the OS. Containers use the namespaces and cgroups mechanisms available in the Linux Kernel to isolate the application from the host environment. They are ideal for micro-services architectures and cloud environments (Cito et al., 2017; Williams, 2016).

[7] `http://flask.pocoo.org`

[8] `https://reactjs.org`

[9] `https://hub.docker.com`

[10] `https://hub.docker.com/r/lelugom/wgs_classifier`

Chapter 4
Experimental Results

I can't be as confident about computer science as I can about biology. Biology easily has 500 years of exciting problems to work on. It's at that level

Donald Knuth
(Apt et al., 2012)

Whole genome sequences of bacteria are currently accessible through publicly available databases of biological sequences. GenBank is the selected database to gather an annotated set of bacterial DNA sequences, as it is part of the International Nucleotide Sequence Database Collaboration and contains Whole Genome Sequence entries. Moreover, GenBank is synchronized daily to the European Nucleotide Archive (ENA) and the DNA Data Bank of Japan (DDBJ), allowing worldwide coverage of genomic sequence information Benson et al. (2012). The database is hosted on the servers of the National Institutes Of Health[1], part of the U.S. Department of Health & Human Services. Daily updates are available through the File Transfer Protocol (FTP) for easy download of the information. After getting the sequence data from GenBank, the dataset is partitioned into training, validation, and test sets for further processing by the recurrent neural network.

First we obtain a recordset for WGS projects – in CSV format – from the NCBI website[2]. Then, the python module performs text processing in the CSV file to extract the WGS project code, which is used to get the project URL of the form `https://www.ncbi.nlm.nih.gov/Traces/wgs/PROJECT_CODE`. Project codes starting with $NZ\_$ do not have a valid HTML page,

[1] `www.ncbi.nlm.nih.gov`

[2] `https://www.ncbi.nlm.nih.gov/Traces/wgs/?page=2&view=wgs&search=BCT`

thus, they are ignored. The python module downloads the project HTML page from the URL and extracts the FTP address and the filename for the compressed FASTA file.

Once the FTP address is extracted, the module proceeds with the download of the compressed FASTA file to the local disk, under a directory with the Taxonomic ID of the species. At most 2000 FASTA compressed files are downloaded per Taxonomic ID. To get Taxonomic ID information, the module downloads the NCBI taxonomy database[3] to the local disk and loads the tables to memory. We filter the information by considering only species with at least 1000 WGS projects (Table 4.1), in order to have sufficient data per class for further neural network training. A rule of thumb in 2016 set the number of samples per class at 5000 for achieving acceptable performance in supervised deep learning models (Goodfellow et al., 2016). However, advances in deep learning architectures, optimization algorithms, and regularization techniques help in the process of using training sets with a lower count of samples per class.

Species	**Tax ID**	**Projects**
Escherichia coli	562	9505
Salmonella enterica	28901	7266
Neisseria meningitidis	487	1314
Listeria monocytogenes	1639	2160
Streptococcus pneumoniae	1313	8318
Mycobacterium tuberculosis	1773	5245
Acinetobacter baumannii	470	2383
Klebsiella pneumoniae	573	3326
Mycobacterium abscessus	36809	1566
Campylobacter jejuni	197	1088
Clostridioides difficile	1496	1199
Shigella sonnei	624	1041
Staphylococcus aureus	1280	8467
Pseudomonas aeruginosa	287	2565

Table 4.1: WGS number of projects per bacteria species with at least one thousand valid entries in the recordset. Species taxonomic ID information included.

The World Health Organization published a priority list for antibiotic-resistant bacteria, taking into account ten criteria: ”all-cause mortality, healthcare and community burden, prevalence of resistance, 10-year trend of resistance, transmissibility, preventability in hospital and community set-

[3] ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxdump.tar.gz

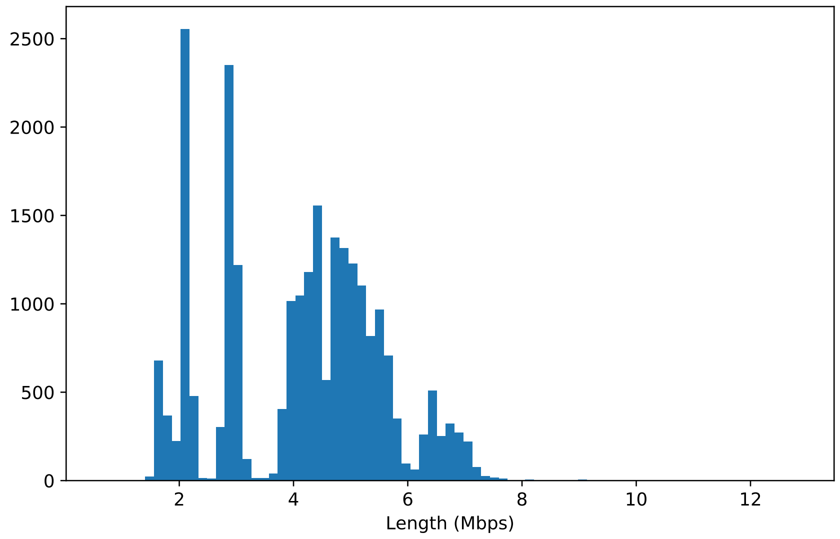


Fig. 4.1: Histogram of sequence lengths for the bacterial whole genome sequences considered in the classification system.

tings, treatability and current pipeline" (Tacconelli and Magrini, 2017). The threshold of a thousand samples per species allows us to cover Mycobacterium tuberculosis – a globally established priority because it is the cause of human tuberculosis, Acinetobacter baumannii, Pseudomonas aeruginosa, and two species from the Enterobacteriaceae group, namely, Klebsiella pneumoniae and Escherichia coli. All of the aforementioned species are part of the critical priority group. We also cover Staphylococcus aureus, Salmonella enterica, and Campylobacter jejuni, which represent half the group of species in the high priority group. From the medium priority group, we cover two of the three species: Streptococcus pneumoniae and Shigella sonnei.

$$bin\ size = \frac{2 * IQR}{\sqrt[3]{n}} \tag{4.1}$$

Using the Freedman and Diaconis rule (Freedman and Diaconis, 1981) in equation 4.1 – where IQR stands for Interquartile Range – we calculate the histogram of the bacterial whole genome sequence lengths (Figure 4.1). The histogram does not include the following abnormal samples, that were also ignored prior to the distributed representation computation:

- Salmonella enterica: 153722611 bps
- Campylobacter jejuni: 118623 bps
- Streptococcus pneumoniae: 270 bps, 340 bps, 666 bps

- Escherichia coli: 127123 bps
- Clostridioides difficile: 38855 bps
- Mycobacterium tuberculosis: 1007 bps, 1547 bps, 73427 bps, 4768 bps

Important metrics for sequence lengths after discarding abnormal samples:

- Minimum: 630582 bps
- Maximum: 12852310 bps
- Mean: 4131930 bps
- Median: 4348813 bps
- Standard deviation: 1435578 bps
- IQR: 2227954 bps

Metrics for sequence lengths facilitate the implementation of the distributed representation computation in C++, where fixed-length arrays are faster than dynamic data structures.

Set	**Percentage (%)**	**No. of sequences**
Training	60	14523
Validation	20	4841
Test	20	4842

Table 4.2: Whole genome sequence set partition into training, testing, and validation sets. The set includes fourteen species.

All the training and evaluation tests were performed using a computer with 8GB of RAM, an Intel Core i7-7700HQ CPU, and a GPU NVidia GeForce GTX 1050 with 2GB of dedicated RAM. The CPU has 4 physical cores and 8 threads through hyperthreading. Its base frequency is 2.8GHz, while the Max Turbo frequency is 3.8GHz. It also has a 6 MB cache with 64-byte cache line length. On the other hand, the GPU is part of the Pascal architecture. It has 640 CUDA cores arranged in 5 streaming multiprocessors. Its clock runs at 1354 MHz, with a boost of 1.3x. Its memory interface width is 128-bits.

Comparisons of tests in computational intelligence need statistical methods to ensure differences are statistically significant. Usually, results from tests are not random samples from a normal distribution. Thus, the assumptions of independence, normality, and homogeneity of variance can not be satisfied. Because of this, it is instrumental to consider nonparametric methods for statistical significance tests. Nonparametric methods are distribution free: their probability distribution function does not change in spite of changes in the sampled population. A number of nonparametric methods have been proposed for pairwise comparisons, including sign test, Wilcoxon test, multiple sign test, and Friedman test (Larsen et al., 2012; Derrac et al., 2011).

The Wilcoxon signed rank test[4] is a well-known nonparametric method. It is less affected by outliers and can be adapted to multiple data structures. This method tests if two samples represent different populations. Thus, it let us know if means from two samples have a significant difference. First, differences between the two samples are calculated. Then, they are ranked by their absolute value. The comparison between the sum of ranks for the positive differences and the sum of ranks for the negative differences gives a T value, which represents the minimum of the two sums. The T value let us know if the null hypothesis – Equation 4.2 from (Larsen et al., 2012) – is rejected and the associated level of significance (p-value) (Larsen et al., 2012; Derrac et al., 2011). We use 5% as the minimum level of significance for the tests throughout this chapter.

$$\begin{aligned} H_0 : \mu_D &= 0 \\ \mu_D &= \mu_X - \mu_Y \end{aligned} \tag{4.2}$$

4.1 Classifiation model tests

The computation of distributed representations is a time consuming process. In order to assess the parallel performance of our implementation, we use the observed speedup, which is calculated as follows:

$$Speedup = \frac{T_s}{T_p} \tag{4.3}$$

where T_s is the wall clock time of serial execution, and T_p is the wall clock time of parallel execution (Barney, 2017). Figure 4.2 depicts the speedup when considering both Python processes and POSIX threads in C++. The sequential time (T_s) considered as a reference is the time that one Python process takes to perform the distributed representation computation. Sixteen samples of the species *Verrucomicrobia bacterium* were considered for the experiment. Computed distributed representations comprised k-mers with $k = \{3, 4, 5\}$.

Speedup results show that the C++ implementation represents a considerable improvement over the Python code. The execution time with 16 threads is 12 times faster than the reference execution time. It is also 3 times faster than the Python implementation with 16 processes. Most of the speedup comes from the optimizations that the C++ compiler performs and the expected improvement that a compiled executable has over an interpreted script. Another factor helping the speedup is the use of constant unordered

[4] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.wilcoxon.html

maps for representing k-mer dictionaries, which are faster than the equivalent dictionary data type in Python.

Once distributed representation computations are finished, we proceed with the training process for the neural network. After dividing the sequences into training, testing, and validation sets (Table 4.2), the softmax cross entropy loss is minimized during the training process. Learning rate was set to 1e-3 and batch size to 128. The Adam optimizer minimized the loss function, with default parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e^{-08}$. Validation set results (Figure 4.3) include an accuracy of 99.0% at 950 steps. Precision was 99.8%, recall 99.93%, and F-score 99.86% at the same step. After finishing training at 3600 steps, results over the test set yielded an accuracy of 99.13%, precision 99.97%, recall 99.91%, and F-score 99.93%. No over-fitting was observed as can be seen when comparing training and validation loss curves.

After analyzing the results of the training process with the validation set, we can realize that losses decrease the first 2800 steps and no further improvement is achieved after 3600 steps. Thus, early stopping (Section 3.3)

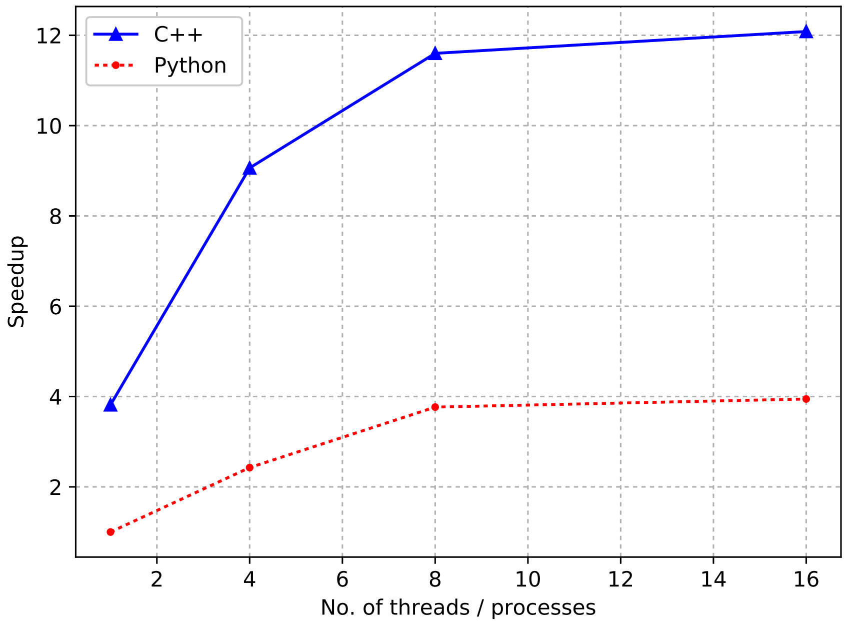


Fig. 4.2: Speedup curves for the distributed representation computations against the number of processes for the Python implementation and the number of threads for the C++ code. The reference time comes from the computation using one Python process.

gives us a range between 2800 and 3600 steps for training the neural network. Our subsequent experiments use a step count in that range.

k-mers	**Accuracy**	**p-value**
$k = \{3\}$	0.97341 ± 0.02960	0.507
$k = \{4\}$	0.98694 ± 0.00751	0.114
$k = \{5\}$	0.93938 ± 0.07192	0.332
$k = \{6\}$	0.92662 ± 0.14023	0.600
$k = \{3, 4\}$	0.98919 ± 0.00512	0.059
$k = \{4, 5\}$	0.97258 ± 0.01852	0.444
$k = \{5, 6\}$	0.97958 ± 0.01337	0.600
$k = \{3, 4, 5\}$	0.97752 ± 0.01594	–
$k = \{4, 5, 6\}$	0.98030 ± 0.01568	0.463

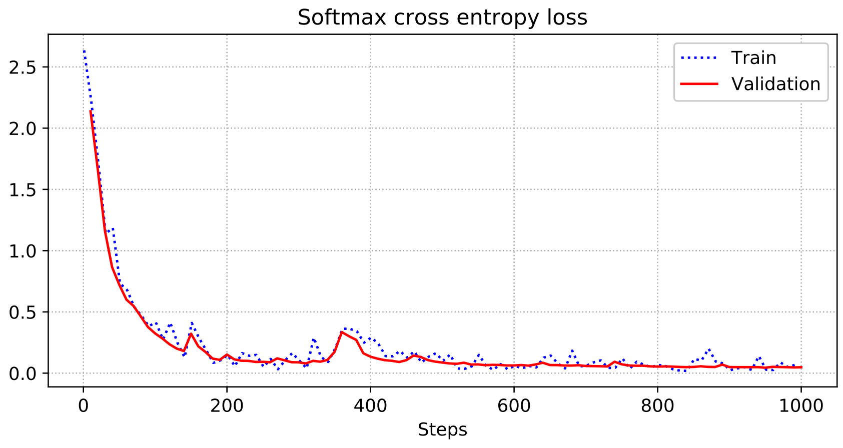
Table 4.3: Influence of k-mers representation on the accuracy of the model. The dataset has fourteen species. k={3,4,5} provides reference accuracy results.

The BRNN is robust to changes in the number of spectral k-mers representations and the number of sequences per species in the dataset. Such results are generated when the dataset has the fourteen classes in table 4.1. Testing with different k-mer configurations yielded no change in the model accuracies calculated over the test set (see table 4.3). The exact same pattern is observed when decreasing the number of sequences per class in the dataset (see table 4.4). No difference has statistical significance according to the p-values obtained.

Because of the robustness against the decrease in sequences per class, it is possible to extend the species coverage of the classification system. Also, results in table 4.4 let us know how to deal with situations in which we have a small number of sequences per class: when we have few sequences per species, we can train the classification system with a small number of classes – fourteen classes in the tests – without impacting the accuracy of the model.

However, in terms of computing resources usage, the distributed representation based on k-mers of $k = \{3, 4, 5\}$ needed around 12GB of RAM, and roughly 30 hours to perform ten training cycles of 3600 steps – without a validation set. When using a validation set, this computing time increases depending on the number of times we stop the training process to validate. Once we change the distributed representation to any combination that includes 6-mers, the memory consumption increases to around 48GB of RAM. The memory increase forces the system to use a lot of swap space, thus, one training cycle with 2800 steps takes about 18 hours to complete.

Regarding hyperparameter search, we performed tests with varying values of learning rates and a different optimizer. Increasing the learning rate



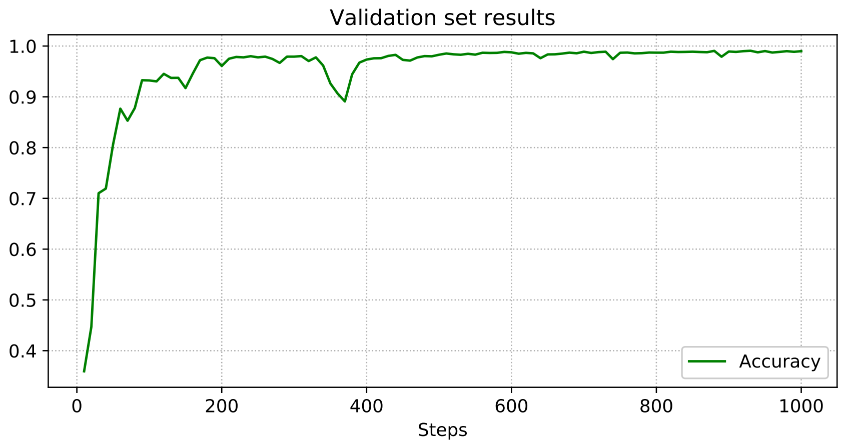


Fig. 4.3: Training curves for the bidirectional recurrent neural network, considering the first one thousand iterations. Image at the top has the loss curves for both training and validation sets. Image at the bottom depicts accuracy evolution over the validation set during the training of the classification system.

Sequences per class	**Accuracy**	**p-value**
1000	0.97792 ± 0.01038	–
750	0.98218 ± 0.00848	0.168
500	0.97484 ± 0.01567	0.507
250	0.97486 ± 0.02368	0.798
100	0.96714 ± 0.01955	0.074

Table 4.4: Variation of accuracy per number of sequences in each class. The dataset has fourteen species.

from $1e^{-3}$ to $1e^{-2}$ generates an accuracy of 0.97958 ± 0.01053 (p-value = 0.345). Although the difference is not statistically significant, a number of divergence errors while training the neural network indicate that this setting is unstable. Thus, we discard this learning rate value. Decreasing the learning rate from $1e^{-3}$ to $1e^{-4}$, without any other change, yields an accuracy of 0.96384 ± 0.03574 (p = 0.172). This change has no significance. On the other hand, changing the Adam optimizer to a Proximal Adagrad Optimizer – with learning rate at $1e^{-3}$ and L_1, L_2 regularization strengths of $1e^{-3}$ – severely affects network metrics. Accuracy drops to 0.72889 ± 0.00851 (p-value = 0.0277). L_1, L_2 regularization strengths of $1e^{-1}$ are even worse: the accuracy drops to 0.06518 ± 0.00175, which is similar to randomly choosing labels for classifying the sequences. Hence, the Adam optimizer (learning rate of $1e^{-3}$) provides the best setting for training the classification system.

Once the training curves let us know that no overfitting affects the model, we set the training steps to 3600 and performed the cross-validation of the model. The experiment comprised ten runs, with randomly selected samples for the training and testing sets of each run. The training set had a proportion of 90% and the testing set had the remaining 10% of the samples. All runs finished at the same training step while each run had different training and testing sets. A heat map with the normalized confusion matrix appears in Figure 4.4. The cross-validation procedure yielded the following metrics: accuracy at 0.98512 ± 0.00798, precision at 0.98542 ± 0.00783, recall at 0.98484 ± 0.00817, and F-score at 0.98513 ± 0.00799. Regarding the accuracy, we got a maximum of 99.586%, which is very similar to the accuracy obtained when training the model with a validation set. However, lower values in other runs decreased the mean accuracy to 98.542%.

Genomic studies might contain partial genomes and sequencing errors could decrease the size of the genome sequences obtained. Therefore, it is important to test the way the model reacts to fragments of whole genome sequences. To perform this test, we randomly choose ten samples for each of the fourteen species. Once the sequences are loaded into memory, we select one fragment per sequence with a random starting position. Then, we compute the distributed representation and standardize each fragment. Percentages

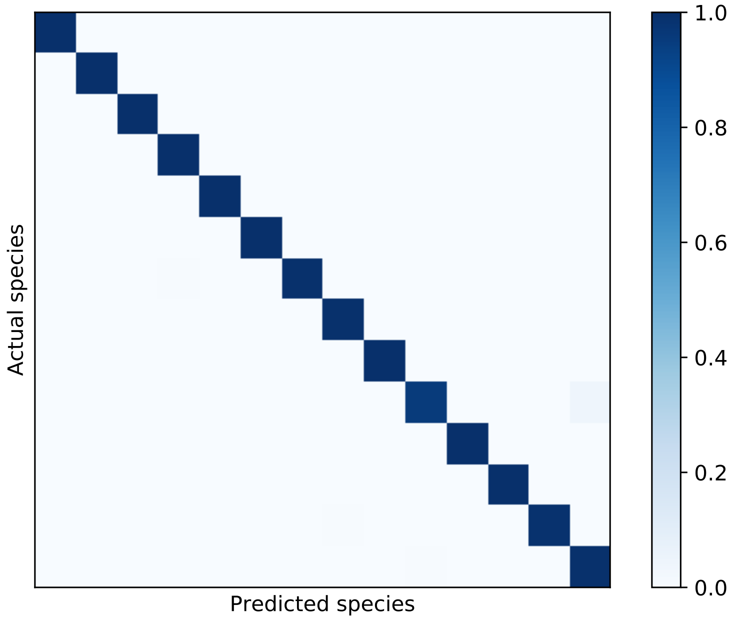


Fig. 4.4: Confusion matrix for the BRNN with fourteen classes. The accuracy of the model is 99.3%.

considered for fragment size include 20%, 40%, 60%, 80%, 90%, 95%, and 100% of the whole sequence. The results generated by the neural network for ten tests appear on figure 4.5.

When considering fragments of 20%, the model failed to identify any species. It generated an accuracy of 0.07143 ± 0.0, the same accuracy that we would get if we randomly choose labels for the test set. The model started to recognize some classes when processing fragments of 40% and 60%. Fragments of 80% improved the results of the classification, yielding an accuracy of 0.80429 ± 0.01895. Fragments of 90% further improved the model accuracy to 0.95571 ± 0.00948. All the differences where statistically significant ($p \leq 0.005$ in all cases). Tests with fragments of 95% yielded an accuracy of 0.98143 ± 0.01069. A p-value of 0.02 implies that the difference has statistical significance against the result obtained without trimming the sequence (0.99286 ± 0.00452). Therefore, the classification system is slightly affected by fragments that represent at least 95% of the original sequence – around 1% drop in accuracy. But smaller fragments have a larger impact on the model accuracy.

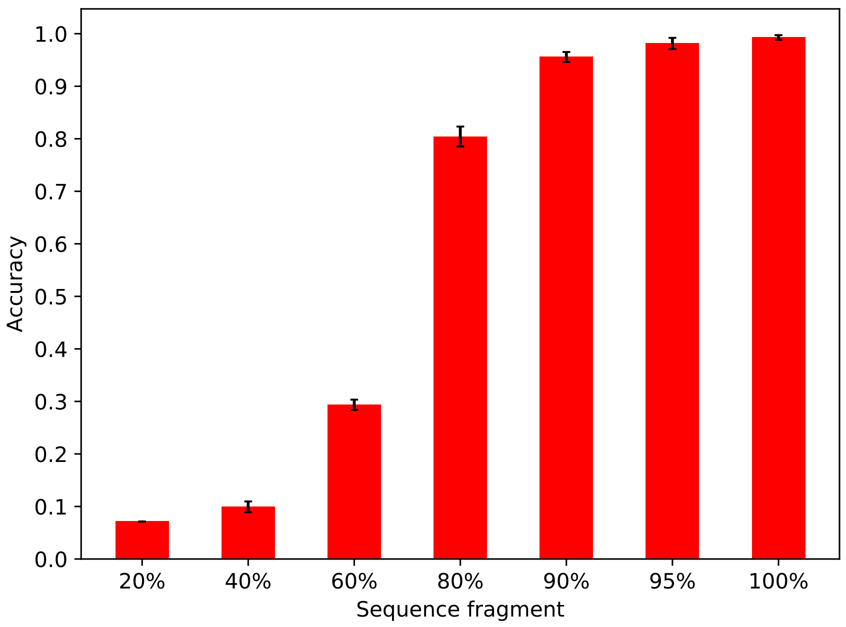


Fig. 4.5: Accuracy generated by the recurrent neural network when classifying fragments of whole genome sequences. The size of the fragment is represented as a percentage of the sequence.

4.2 Species coverage increase

Results in table 4.4 represent tests with different values for the sequences per class. However, all the tests use a fixed number of fourteen classes (Table 4.1). Given the robustness to variations in sequences per class, we extend the coverage of the classification system by lowering the threshold for the minimum samples per species. Because of this, we repeat the tests, changing the thresholds and increasing the species counts. Every test is repeated ten times, randomly selecting 80% of the samples for training the neural network, and the remaining 20% of the samples for testing.

Model accuracies decrease as the number of sequences per class decreases (see Table 4.5). Although the change from 1000 sequences per class to 750 sequences per class has no statistically significant difference (both unbalanced and manually balanced tests yield p values above 0.05), the remaining decreases in minimum sequences per class do generate statistically significant drops in accuracy results. It is important to highlight that unbalanced and manually balanced datasets do not generate statistically significant differences when minimum sequences are set to 1000, 750, and 500. The robustness

Dataset	Sequences	Accuracy	p-value	Species
Unbalanced	1000	0.97752 ± 0.01594	–	14
	750	0.94172 ± 0.05347	0.168	22
	500	0.94255 ± 0.04499	0.036	27
	250	0.86609 ± 0.03144	0.005	48
	100	0.76981 ± 0.08550	0.005	111
Balanced	1000	0.97792 ± 0.01038	–	14
	750	0.94438 ± 0.04427	0.092	22
	500	0.92002 ± 0.04684	0.028	27
	250	0.76266 ± 0.08721	0.005	48
	100	0.68913 ± 0.08310	0.005	111

Table 4.5: Variation of accuracy against minimum sequences per species. Tests were repeated ten times. Training set 80% and testing set 20%.

of neural networks to unbalanced datasets explains this results. However, minimum sequences of 250 and 100 per species represent statistically significant differences between unbalanced and manually balanced datasets. Differences in these two cases are explained by the reduction in the dataset size when conducting the manual balancing.

Sequences	Unbalanced	Balanced	p-value
1000	0.97752 ± 0.01594	0.97792 ± 0.01038	0.721
750	0.94172 ± 0.05347	0.94438 ± 0.04427	0.798
500	0.94255 ± 0.04499	0.92002 ± 0.04684	0.444
250	0.86609 ± 0.03144	0.76266 ± 0.08721	0.028
100	0.76981 ± 0.08550	0.68913 ± 0.08310	0.046

Table 4.6: Variation of accuracy against minimum sequences per species. Comparison between the unbalanced dataset and a manually balanced dataset.

We now set the dataset to a minimum of 250 sequences per class to test subsequent changes in the model architecture. It is the first threshold that makes the accuracy drop below 90% (Table 4.6), so we want to improve the neural network architecture performance.

The first test assessed the addition of a dense module (Park et al., 2017) on top of the bidirectional recurrent layer. The dense module takes the concatenation of the last bidirectional state vector and the last bidirectional output vector. It then processes the concatenated output with two dense layers – also known as fully connected layers – with *tanh* as an activation function. Results are then fed to the classification layer. The addition of the dense module has a positive impact on the model: the accuracy increases from 0.86609 ± 0.03144 to 0.91926 ± 0.03614, with a p-value of 0.028.

Then, we returned to the base model and added the attention mechanism (Bahdanau et al., 2015) using the TensorFlow implementation[5] (Section 3.4). Results represent a considerable improvement over the base model. The accuracy increases from 0.86609 ± 0.03144 to 0.95031 ± 0.00581 (p-value $= 0.027$). Hence, a context vector consolidating all intermediate results adds valuable information to the last output vector. Weights for context vector calculation allows the model to learn what are the most important intermediate vectors to improve the final classification result.

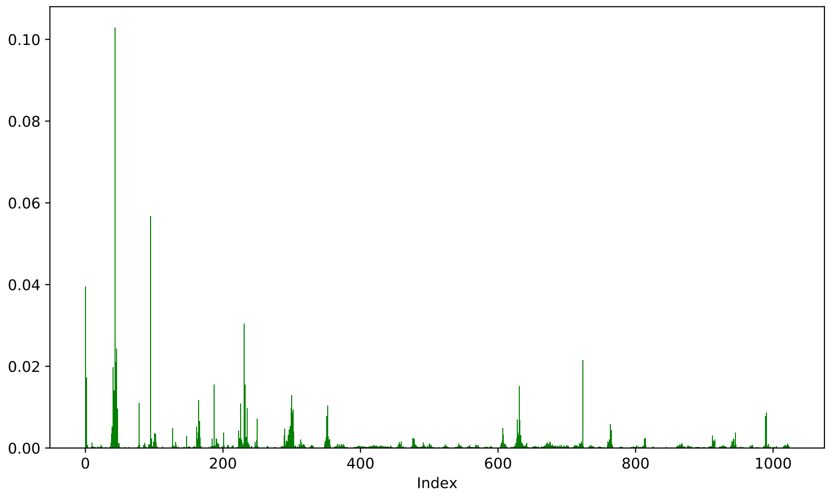


Fig. 4.6: Mean values for the alignment values generated by the attention mechanism (Bahdanau et al., 2015). The highest mean values are related to inputs in the first part of the distributed representation.

An analysis of the weights for context vector calculation – also known as alignments – provides additional insight into the sequence representations. After computing the mean values for the alignments generated for the sequences in the test set (Figure 4.6), we can observe that the four most critical intermediate results are related to inputs in the first part of the sequence representation. The first part of the dataset entry contains values for $k = \{3, 4, 5\}$ and $k = \{4, 5\}$, a significant result that highlights the importance of the distributed representation for generating the final classification

[5] `https://github.com/tensorflow/tensorflow/blob/r1.7/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py`

output. Alignments mean values also help to explain the improvement that the context vector represents when concatenating it to the last output vector. They cover intermediate results for the whole dataset entry.

A comparison between scoring functions from attention mechanisms in (Luong et al., 2015) and (Bahdanau et al., 2015) did not produce a statistically significant difference. The scoring function in (Luong et al., 2015) generated an accuracy of 0.94885 ± 0.00294, with a p-value of 0.027 against the base model. But a p-value of 0.345 when comparing both scoring function variants implies that they generate equivalent results for our classification system.

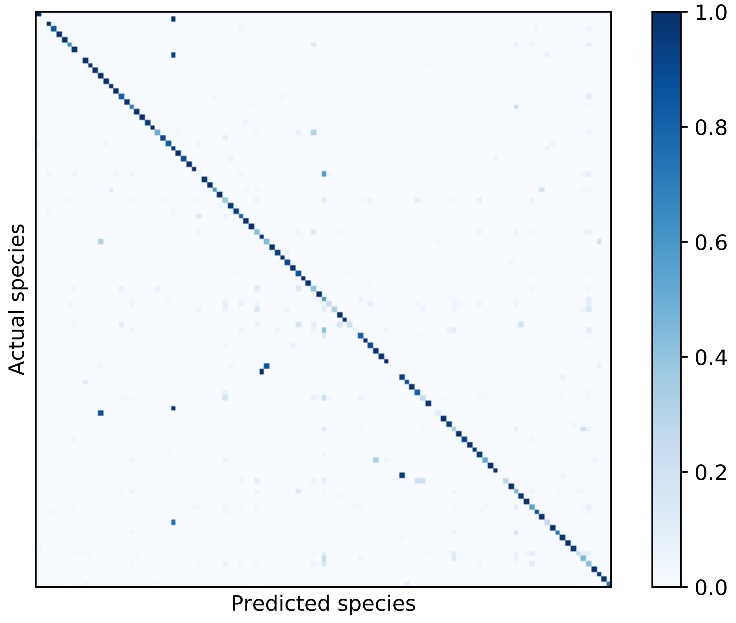


Fig. 4.7: Confusion matrix for the final classification model covering 111 species. The accuracy of the model is 89.632%.

Probabilities generated by the softmax layer let us know the confidence of the resulting species labels. Those probabilities represent a score for each neural network inference. Also, they can ease the identification of species that were not part of the training set. To determine the score threshold, we trained the system with a minimum of two hundred samples per species – 48 species. The resulting scores from the testing set had a mean of 0.95476 and a variance of 0.01636. Then, we build a test set with a minimum of one hundred samples per species – 111 species. For this dataset, the resulting scores had a mean of 0.91748 and a variance of 0.03031. Thus, we set the score threshold to be 0.9384, the middle point between the two means. Probabilities under this threshold express low confidence in the resulting species label.

Although the dense module and the attention mechanism improved the classification system by their own, their combination ended up harming the model. Accuracy yielded 0.93345 ± 0.00890 (p-value = 0.027), a result between dense module and attention mechanism accuracies. An explanation for this behavior can be found in the difference between trainable parameters count. When combining both mechanisms, there is an increase in neural network weights. But there is no increase in the number of sequences per species. Thus, model training becomes more difficult. However, the decrease in model performance when we add more parameters is a good indicator. Usually, the best models have the largest amount of trainable parameters that the available dataset size permits. Hence, our model is at the limit for the given set of training examples.

Sequences	**Base model**	**Final model**	**p-value**
1000	0.97752 ± 0.01594	0.99455 ± 0.00281	0.005
500	0.94255 ± 0.04499	0.98716 ± 0.01263	0.027
250	0.86609 ± 0.03144	0.95031 ± 0.00469	0.027
100	0.76981 ± 0.08550	0.89039 ± 0.00417	0.006

Table 4.7: Variation of accuracy against minimum sequences per species. Comparison between the base model and the final model.

The following changes did not produce any positive effect on the architecture: weighted loss, Glorot uniform initialization, layer size decrease, and dropout wrappers for the GRU cells. In the weighted loss test, inverse frequency provided the weights to correct for species imbalance (Katevas et al., 2017). Results showed a drop in accuracy to 0.92366 ± 0.00894. The Glorot uniform initialization (Glorot and Bengio, 2010) for neural network weights generated an accuracy of 0.94042 ± 0.008. A layer size decrease from 128 to 64 GRU cells made accuracy drop to 0.93769 ± 0.00581. The dropout wrappers (Gal and Ghahramani, 2016) caused a considerable reduction in performance: accuracy yielded 0.70061 ± 0.00524. All results were statistically significant against the model with the attention mechanism (accuracy of 0.95031 ± 0.00469).

Therefore, our final model has a bidirectional GRU layer with 128 units plus the attention mechanism (Bahdanau et al., 2015). A comparison between the base model and the final model for different thresholds of minimum sequences appear in Table 4.7. All results are statistically significant. We can observe a substantial improvement against the base model in all the scenarios, although the lower the threshold, the higher the value of the final model improvement. With a threshold of one hundred sequences per species, the model achieves the broadest coverage. A confusion matrix with this threshold (Figure 4.7) let us know the main limitation of the model: It is clear from the matrix diagonal how the model struggles to classify a few species, affecting the overall accuracy of the architecture.

4.3 Validation of results

Now that we have all the results from the experiments with the recurrent neural network, we select alternative methods for results comparison. Naive Bayes (NB) and Multilayer Perceptron (MLP) are two classification models which provide a reference to compare our classification system results.

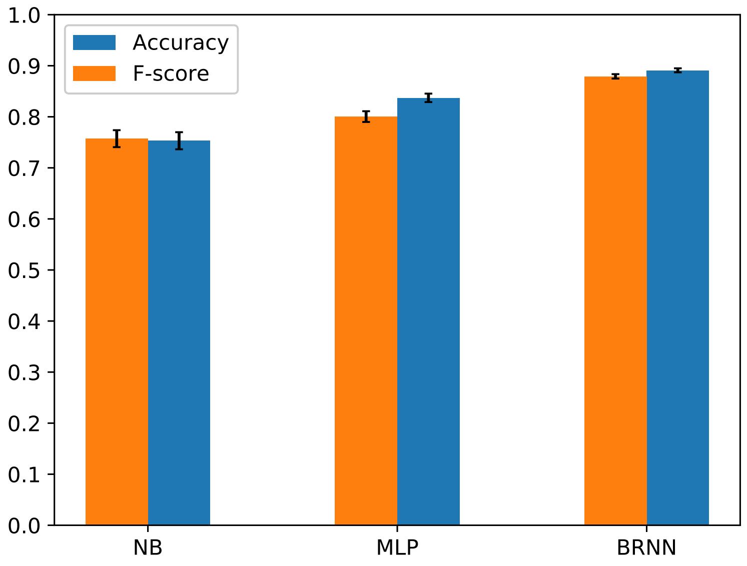


Fig. 4.8: Validation of results comparing Naive Bayes (NB), Multilayer Perceptron (MLP), and our classification model.

Naive Bayes is a simple yet powerful classifier. An important advantage of this method is the lack of hyperparameters to adjust. The classifier is based on the Bayes theorem (Alpaydin, 2014):

$$P(C|\mathbf{x}) = \frac{P(C)p(\mathbf{x}|C)}{p(\mathbf{x})} \tag{4.4}$$

where $P(C)$ is the prior probability, $p(\mathbf{x}|C)$ is the class likelihood, and $p(\mathbf{x})$ is the evidence. Its basic assumption is that dimensions of input vectors are independent. However, the Naive Bayes classifier yields good performance even when the assumption is not satisfied. The method fits a Gaussian distribution to represent each class in the training dataset. As dimensions in the input vector are assumed to be independent, the covariance matrix is diagonal. The Naive Bayes classifier generates the result by selecting the class with the highest posterior probability (Equation 4.5) for the given input sample (Alpaydin, 2014; Christopher, 2006).

$$P(C_i|\mathbf{x}) = \frac{p(\mathbf{x}|C_i)P(C_i)}{\sum_{k=1}^{K} p(\mathbf{x}|C_k)P(C_k)} \tag{4.5}$$

For the MLP model (Section 2.5), we select the reference implementation of a DNN classifier in TensorFlow[6], with default parameters (ReLU activation, Adagrad optimizer). A dropout rate of 0.5 provided regularization to the model – the same dropout rate the recurrent neural network uses. The model has three hidden layers with the following units: $\{1344, 512, 256\}$. We use 3600 iterations for training the model and a batch size of 128. Default parameters yielded a very low accuracy (0.57181 ± 0.15597). Thus, we changed the MLP model to use tanh activation and Adam optimizer. It's important to note that we can not use a matrix representation (3x1024) for Naive Bayes and MLP models. Instead, we encode data with the concatenated representation (1x1344) that holds k-mers of $k = \{3, 4, 5\}$ (Section 3.2).

We performed cross-validation with 90% of the sequences for training and 10% for testing. Training cycles were repeated ten times. The dataset has a threshold of at least one hundred sequences per species (See Appendix A for a list of species). Such threshold provides the major coverage of all tested scenarios. Results show that our classification model (BRNN GRU) outperforms other machine learning methods such as NP and MLP (Figure 4.8, Table 4.8). There is a considerable increase in performance when considering all the metrics used for model characterization. All the differences in NB and MLP metrics against our classification model were statistically significant.

Other recurrent neural models provided additional means for validating results. Cross-validation tests with a model based on LSTM units and a unidirectional architecture allowed to compare the model performance against common recurrent architectures. For the unidirectional architectures, we doubled the number of units to perform a fair comparison to our classification model. Results in Table 4.8 exhibited similar outputs for all the recurrent neural network configurations.

[6] https://www.tensorflow.org/api_docs/python/tf/estimator/DNNClassifier

Model	**Metric**	**Value**
NB	Accuracy	0.75313 ± 0.01675
	Precision	0.83775 ± 0.00461
	Recall	0.69128 ± 0.02774
	F-score	0.75719 ± 0.01662
MLP	Accuracy	0.83713 ± 0.00829
	Precision	0.79688 ± 0.01054
	Recall	0.80402 ± 0.01308
	F-score	0.80040 ± 0.01055
RNN GRU	Accuracy	0.88896 ± 0.00367
	Precision	0.87889 ± 0.00293
	Recall	0.87354 ± 0.00491
	F-score	0.87620 ± 0.00354
BRNN LSTM	Accuracy	0.89690 ± 0.00153
	Precision	0.89123 ± 0.00594
	Recall	0.88329 ± 0.00192
	F-score	0.88724 ± 0.00334
BRNN GRU	Accuracy	0.89107 ± 0.00392
	Precision	0.88068 ± 0.00515
	Recall	0.87767 ± 0.00486
	F-score	0.87917 ± 0.00436

Table 4.8: Comparison between Naive Bayes (NB), Multilayer Perceptron (MLP), alternative RNN configurations, and our classification model (BRNN GRU).

Chapter 5
Conclusions

But as the term pure suggests, many researchers value such research more than they do applied. They believe that the pursuit of knowledge "for its own sake" reflects humanity's highest calling — to know more, not for the sake of money or power, but for the transcendental good of greater understanding and a richer life of the mind

(Booth et al., 2008)

Although a number of methods are available for the essential task of bacteria identification – including mass spectrometry, pairwise sequence comparison, and microscopic morphology – recurrent neural networks represent an automatic classification method which does not require any manual feature extraction. They are easily updated through the retraining of the model. Our classification system exploits the vast amounts of genomic information available in GenBank to infer the species of a given bacterial whole genome sequence. GenBank provides the samples to train and test the prediction capabilities of our neural network architecture. The model has the potential to benefit diverse areas, such as pathology, microbiology, experimental biology, food and water industries, and evolutionary studies.

A distributed representation provides an excellent encoding for the bacterial genomic information in a low dimensional space. This is an important aspect considering the high dimensionality and sparsity of one-hot encoding sequence representations. The combination of two or more k-mer lengths gives context to the distributed representation. Context takes advantage of positional information, a crucial aspect in biological sequences. From applications in Natural Language Processing, the additional context proved to be useful for our classification model efficiency. On top of that, the distributed representation does not require the assumption that nodes in the FASTA files are ordered.

Our base model has 128 GRU gates arranged in a bidirectional configuration. The classification layer uses softmax directly over the concatenated output state of the bidirectional recurrent layer. Dropout of 0.5 provides regularization to the model before feeding the softmax layer. With a minimum of one thousand sequences per species (14 species), the best accuracy is 99.586%. Cross-validation results generated an accuracy of 0.98512 ± 0.00798. The threshold of one thousand sequences per species allows the coverage of the four critical bacterial species according to the WHO.

Another important test was the response of the network to variations in the size of fragments extracted from the sequence. We found that the model can withstand fragments representing 95% of the original sequences. However, smaller fragments do have a negative impact on the model, generating a drop in accuracy from 0.99286 ± 0.00452 to 0.80429 ± 0.01895 when the fragment size represents only 80% of the original sequence.

Increasing the species coverage made the base model accuracy drop to 0.76981 ± 0.08550 – with a minimum of one hundred sequences per species (111 species). Hence, we iterated over the base model, testing architecture and hyperparameter modifications. The final model includes a bidirectional recurrent layer with 128 GRU cells, a global attention mechanism – with a dense layer to concatenate the context vector to the final output state – and a dense layer for classification. Dropout of 0.5 provides regularization for the two dense layers. Also, the Adam optimizer with a learning rate of 1e-3 minimizes the softmax cross entropy loss function.

Starting from a base neural network model provides an efficient approach for exploring existing proposed models and leveraging experimental results from existing publications. Once the base model is tested, further experiments can help to find correct hyperparameters and architecture variants for the base model. Iterating over the base model allows the improvement of the classification architecture because iterations target its deficiencies and limitations. Such methodology enables a fast and practical search for an optimal neural network model by directing experiments on model improvements.

The classification system leverages context in two different parts of the model: the preprocessing for sequence representation computation and the intermediate processing before the classification layer. The encoding that represents the sequences embeds context in the form of a distributed representation. Also, the attention mechanism – on top of the bidirectional recurrent layer – takes advantage of context from the intermediate outputs of the GRUs. The additional context of the attention mechanism provides a significant improvement over the base model metrics, creating a positive impact on the model accuracy. Such improvement is more noticeable when the sequence minimum count decreases.

Our bidirectional GRU model outperformed other machine learning methods in the classification of bacterial whole genome sequences. Results show better numbers in all the metrics used to characterize the model (accuracy, precision, recall, and balanced F-score). Alternative recurrent neural network architectures – standard GRUs, bidirectional LSTMs – provided equivalent results, with no statistically significant differences.

5.1 Future work

As more curated samples of whole genome sequences are available at GenBank, the classification system can improve in two aspects. First, the number of species with at least one hundred sequences will increase, growing the coverage of the recurrent neural network. Secondly, the accuracy of the system has the potential to improve because most of the species that the model struggles to identify have low sequence counts. Also, proper data augmentation techniques could increase the number of sequences for training. Those augmentation techniques should avoid direct modifications of the nucleotides in the whole genome sequences. Nucleotide changes insert mutations in the sequences. Mutations in core parts of the sequence express alterations in vital functions, which generate a sequence that can not represent a living organism.

Should more computing power is available, evolutionary computation approaches can help in the search for model hyperparameters and neural network architectures. Indeed, evolutionary algorithms offer an interesting approach to automate the search for machine learning architectures (Lipton et al., 2015). Evolutionary algorithms are more effective than grid or random search methods. They can efficiently perform massive parallel searches in high dimensional spaces. Because of this, the hyperparameters and the architecture of the bacterial identification system can be further improved.

The use of Field Programmable Gate Array (FPGA) chips for high throughput and large dimensional data processing could be an important contribution for parallel architectures in deep neural networks. Although design with these architectures moves the machine learning models to the hardware front, FPGAs are particularly suited for those parallel systems. Arrays of FPGAs can efficiently process lots of data in a distributed manner, and their reconfiguration capabilities can enable fast design and prototyping. Comparing them with GPU and CPU implementations, they can represent an improvement regarding performance and power consumption (Farabet et al., 2011).

Application-specific integrated circuits (ASICs) represent another improvement in the hardware front, as gains in cost, energy, and performance are believed to come from domain-specific hardware (Jouppi et al., 2017).

ASICs have been giving promising results regarding performance and power consumption. By performing computations directly in the memory, new chip designs have achieved considerable improvements regarding training times and energy consumption (Hardesty, 2018). This enables the execution of neural networks in embedded devices – usually limited in terms of processing power and energy consumption – opening the possibility to integrate prediction models in NGS hardware. Neural network training and inference in data centers are fundamental as well. TPUs are custom chips designed for data centers. They are faster and more power efficient than their contemporary GPUs and CPUs. Also, cloud versions of TPUs are now accessible in beta for users (Barrus, 2018; Jouppi et al., 2017).

Appendix A
Species list

Species	Tax ID	Projects
Lactobacillus rhamnosus	47715	114
Campylobacter jejuni	197	1088
Microbacterium sp.	51671	112
Streptococcus sp.	1306	151
Streptomyces sp.	1931	539
Enterobacter cloacae	550	645
Staphylococcus epidermidis	1282	498
Pseudomonas syringae	317	311
Mycobacterium tuberculosis	1773	5245
Salmonella enterica	28901	7266
Vibrio cholerae	666	765
Escherichia coli	562	9505
Klebsiella pneumoniae	573	3326
Mycobacterium abscessus	36809	1566
Acinetobacter baumannii	470	2383
Acinetobacter sp.	472	233
Brucella abortus	235	161
Bacillus cereus	1396	956
Enterococcus faecium	1352	796
Enterococcus faecalis	1351	541
Pseudomonas aeruginosa	287	2565
Serratia marcescens	615	343
Burkholderia pseudomallei	28450	657
Neisseria gonorrhoeae	485	438
Pseudomonas sp.	306	727
Staphylococcus aureus	1280	8467
Streptococcus agalactiae	1311	905

Rhizobium sp.	391	121
Sphingomonas sp.	28214	124
Lachnospiraceae bacterium	1898203	298
Clostridiales bacterium	1898207	239
Yersinia pestis	632	289
Streptococcus pyogenes	1314	301
Burkholderia sp.	36773	126
Mesorhizobium sp.	1871066	104
Proteobacteria bacterium	1977087	154
Verrucomicrobia bacterium	2026799	129
Firmicutes bacterium	1879010	232
Mycobacterium sp.	1785	190
Prevotella sp.	59823	221
Xanthomonas oryzae	347	122
Bacillus subtilis	1423	102
Lactobacillus plantarum	1590	201
Helicobacter pylori	210	751
Bordetella pertussis	520	326
Vibrio parahaemolyticus	670	811
Bacillus thuringiensis	1428	437
Stenotrophomonas maltophilia	40324	325
Oenococcus oeni	1247	216
Gammaproteobacteria bacterium	1913989	446
Listeria monocytogenes	1639	2160
Staphylococcus sp.	29387	207
Shigella flexneri	623	171
Bacillus anthracis	1392	162
Lactococcus lactis	1358	108
Clostridium botulinum	1491	178
Clostridioides difficile	1496	1199
Clostridium sp.	1506	144
Bacteroides fragilis	817	115
Legionella pneumophila	446	490
Rhodococcus sp.	1831	119
Paenibacillus sp.	58172	129
Cronobacter sakazakii	28141	171
Vibrio sp.	678	115
Pseudomonas stutzeri	316	225
Acinetobacter pittii	48296	156
Acidobacteria bacterium	1978231	110

Shigella sonnei	624	1041
Bacteroidetes bacterium	1898104	196
Streptococcus pneumoniae	1313	8318
Streptococcus equi	1336	246
Francisella tularensis	263	186
Corynebacterium diphtheriae	1717	183
Mycobacterium avium	1764	182
Neisseria meningitidis	487	1314
Bacteroidales bacterium	2030927	464
Actinobacteria bacterium	1883427	176
Vibrio vulnificus	672	104
Klebsiella oxytoca	571	106
Yersinia enterocolitica	630	162
Porphyromonadaceae bacterium	2049046	176
Streptococcus suis	1307	989
Streptococcus mutans	1309	180
Klebsiella aerogenes	548	133
Campylobacter coli	195	795
Klebsiella quasipneumoniae	1463165	136
Burkholderia cenocepacia	95486	239
Pasteurella multocida	747	116
Corynebacterium sp.	1720	128
Deltaproteobacteria bacterium	2026735	259
Euryarchaeota archaeon	2026739	487
Chloroflexi bacterium	2026724	256
Enterobacter hormaechei	158836	377
Bacillus pseudomycoides	64104	104
Haemophilus influenzae	727	130
Ruminococcaceae bacterium	1898205	195
Leptospira interrogans	173	286
Burkholderia ubonensis	101571	288
Campylobacter concisus	199	123
Flavobacteriaceae bacterium	1871037	254
Alphaproteobacteria bacterium	1913988	231
Flavobacteriales bacterium	2021391	270
Bacillales bacterium	1904864	137
Dehalococcoidia bacterium	2026734	112
Prochlorococcus sp.	1220	151
Staphylococcus haemolyticus	1283	175
Staphylococcus argenteus	985002	110
Elusimicrobia bacterium	2030800	101

Rhodospirillaceae bacterium	1898112	104
Bacillus toyonensis	155322	201
Bacillus wiedmannii	1890302	131

Table A.1: WGS number of projects per bacteria species with at least one hundred valid entries in the recordset. Species taxonomic ID information included.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*.

Alahi, A., Ortiz, R., and Vandergheynst, P. (2012). FREAK: Fast retina keypoint. In *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*, pages 510–517. IEEE.

Alipanahi, B., Delong, A., Weirauch, M. T., and Frey, B. J. (2015). Predicting the sequence specificities of DNA -and RNA- binding proteins by deep learning. *Nature biotechnology*, 33(8):831–838.

Alpaydin, E. (2014). *Introduction to machine learning.* MIT press, Cambridge, MA, US.

Angermueller, C., Pärnamaa, T., Parts, L., and Stegle, O. (2016). Deep learning for computational biology. *Molecular systems biology*, 12(7):878.

Apt, K. R., Marek, V. W., Truszczynski, M., and Warren, D. S. (2012). *The Logic Programming Paradigm: A 25-Year Perspective.* Springer Science & Business Media.

Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D., and Nieh, J. (2016). POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 19. ACM.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. (2015). Comparative study of deep learning software frameworks. *arXiv preprint* *arXiv:1511.06435.*

Barney, B. (2017). High Performance Computing. https://computing.llnl.gov/tutorials/.

Barrus, J. (2018). Cloud TPU machine learning accelerators now available in beta. https://cloudplatform.googleblog.com/2018/02/Cloud-TPU-machine-learning-accelerators-now-available-in-beta.html.

Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., and Sayers, E. W. (2012). Genbank. *Nucleic acids research*, 41(D1):D36–D42.

Booth, W. C., Colomb, G. G., and Williams, J. M. (2008). *The craft of research.* University of Chicago press.

Bosco, G. L. and Di Gangi, M. A. (2016). Deep Learning Architectures for DNA Sequence Classification. In *International Workshop on Fuzzy Logic and Applications*, pages 162–171. Springer.

Bradski, G. and Kaehler, A. (2008). *Learning OpenCV: Computer vision with the OpenCV library.* O'Reilly Media, Inc.

Cheng, S., Guo, M., Wang, C., Liu, X., Liu, Y., and Wu, X. (2016). MiRTDL: a deep learning approach for miRNA target prediction. *IEEE/ACM transactions on computational biology and bioinformatics*, 13(6):1161–1169.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014a). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078.*

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, abs/1406.1078.

Christopher, M. B. (2006). *Pattern recognition and machine learning.* Springer-Verlag New York.

Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., and Gall, H. C. (2017). An empirical analysis of the Docker container ecosystem on GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 323–333. IEEE Press.

Cole, J. R., Wang, Q., Fish, J. A., Chai, B., McGarrell, D. M., Sun, Y., Brown, C. T., Porras-Alfaro, A., Kuske, C. R., and Tiedje, J. M. (2013). Ribosomal database project: data and tools for high throughput rrna analysis. *Nucleic acids research*, 42(D1):D633–D642.

Dahl, G. E. (2015). *Deep learning approaches to problems in speech recognition, computational chemistry, and natural language text processing.* PhD thesis, University of Toronto.

Derrac, J., García, S., Molina, D., and Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18.

Desolneux, A., Moisan, L., and Morel, J.-M. (2007). *From gestalt theory to image analysis: a probabilistic approach*, volume 34. Springer Science & Business Media.

Dolz, J., Desrosiers, C., and Ayed, I. B. (2016). 3D fully convolutional networks for subcortical segmentation in MRI: A large-scale study. *arXiv preprint arXiv:1612.03925*, abs/1612.03925.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Efron, B. and Hastie, T. (2016). *Computer age statistical inference. Algorithms, Evidence, and Data Science.* Cambridge University Press.

Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., and Talay, S. (2011). Large-scale fpga-based convolutional networks. *Scaling up Machine Learning: Parallel and Distributed Approaches*, pages 399–419.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37.

Forrest, S. and Mitchell, M. (2016). Adaptive computation: the multidisciplinary legacy of john h. holland. *Communications of the ACM*, 59(8):58–63.

Freedman, D. and Diaconis, P. (1981). On the histogram as a density estimator: L 2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57(4):453–476.

Gal, Y. and Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027.

Garrity, G. M. (2016). A New Genomics-Driven Taxonomy of Bacteria and Archaea: Are We There Yet? *Journal of clinical microbiology*, 54(8):1956–1963.

Giang Nguyen, N., Tran, V. A., Ngo, D. L., Phan, D., Lumbanraja, F. R., Faisal, M. R., Abapihi, B., Kubo, M., and Satou, K. (2016). DNA Sequence Classification by Convolutional Neural Network. *J. Biomedical Science and Engineering*, 9(9):280–286.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press, Cambridge, MA, US.

Graves, A. (2012). *Supervised sequence labelling with recurrent neural networks.* PhD thesis.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850.*

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401.*

Hardesty, L. (2018). Neural networks everywhere. http://news.mit.edu/2018/chip-neural-networks-battery-powered-devices-0214.

Harris, M. (2017). An Even Easier Introduction to CUDA. https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/.

Hasman, H., Saputra, D., Sicheritz-Ponten, T., Lund, O., Svendsen, C. A., Frimodt-Møller, N., and Aarestrup, F. M. (2013). Rapid whole genome sequencing for the detection and characterization of microorganisms directly from clinical samples. *Journal of clinical microbiology*, 52(1).

Hassaballah, M., Abdelmgeid, A. A., and Alshazly, H. A. (2016). Image features detection, description and matching. In *Image Feature Detectors and Descriptors*, pages 11–45. Springer.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hyötyniemi, H. (1996). Turing machines are recurrent neural networks. pages 13–24. STeP '96 - Genes, Nets and Symbols; Finnish Artificial Intelligence Conference, Vaasa, Finland, 20-23 August 1996.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350.

Katevas, K., Leontiadis, I., Pielot, M., and Serrà, J. (2017). Practical processing of mobile sensor data for continual deep learning predictions. *arXiv preprint arXiv:1705.06224.*

Kimothi, D., Soni, A., Biyani, P., and Hogan, J. M. (2016). Distributed representations for biological sequence analysis. *arXiv preprint arXiv:1608.05949.*

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

Krawczyk, P. S., Lipinski, L., and Dziembowski, A. (2018). PlasFlow: predicting plasmid sequences in metagenomic data using genome signatures. *Nucleic acids research.*

Lanchantin, J., Singh, R., Wang, B., and Qi, Y. (2016). Deep motif dashboard: Visualizing and understanding genomic sequences using deep neural networks. *arXiv preprint arXiv:1608.03644.*

Larsen, R. J., Marx, M. L., et al. (2012). *An introduction to mathematical statistics and its applications.* Pearson, fifth edition.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lee, B., Baek, J., Park, S., and Yoon, S. (2016). deepTarget: end-to-end learning framework for microRNA target prediction using deep recurrent neural networks. *arXiv preprint arXiv:1603.09123*.

Lee, T. K. and Nguyen, T. (2016). Protein family classification with neural networks. https://cs224d.stanford.edu/reports/LeeNguyen.pdf.

Leekitcharoenphon, P., Kaas, R. S., Thomsen, M. C. F., Friis, C., Rasmussen, S., and Aarestrup, F. M. (2012). snpTree-a web-server to identify and construct SNP trees from whole genome sequence data. In *BMC genomics*, volume 13, page S6. BioMed Central.

Lehman, E., Leighton, F. T., and Meyer, A. R. (2010). Mathematics for computer science. Technical report, Technical report. Lecture notes.

Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

Liu, H., Wang, Z., Shen, B., and Alsaadi, F. E. (2016). State estimation for discrete-time memristive recurrent neural networks with stochastic time-delays. *International Journal of General Systems*, 45(5):633–647.

Liu, X. (2017). Deep recurrent neural network for protein function prediction from sequence. *arXiv preprint arXiv:1701.08318*.

Lodish, H., Berk, A., Zipursky, S. L., Matsudaira, P., Krieger, M., Darnell, J., et al. (2004). *Molecular cell biology*. W.H. Freeman and CO, New York, US, fifth edition.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Markowitz, V. M., Chen, I.-M. A., Palaniappan, K., Chu, K., Szeto, E., Grechkin, Y., Ratner, A., Jacob, B., Huang, J., Williams, P., et al. (2011). Img: the integrated microbial genomes database and comparative analysis system. *Nucleic acids research*, 40(D1):D115–D122.

Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040. Citeseer.

Meyer, F., Paarmann, D., D'Souza, M., Olson, R., Glass, E. M., Kubal, M., Paczian, T., Rodriguez, A., Stevens, R., Wilke, A., et al. (2008). The metagenomics RAST server–a public resource for the automatic phylogenetic and functional analysis of metagenomes. *BMC bioinformatics*, 9(1):386.

Min, S., Lee, B., and Yoon, S. (2016). Deep learning in bioinformatics. *arXiv preprint arXiv:1603.06430*, abs/1603.06430.

Mitchell, M. (1995). Genetic algorithms: An overview. *Complexity*, 1(1):31–39.

Mohamad, N. A., Jusoh, N. A., Htike, Z. Z., and Win, S. L. (2014). Bacteria identification from microscopic morphology: a survey. *International Jour-*

nal on Soft Computing, Artificial Intelligence and Applications (IJSCAI), 3(1):2319–1015.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective.* The MIT Press, Cambridge, MA, US.

Ng, P. (2017). dna2vec: Consistent vector representations of variable-length k-mers. *arXiv preprint arXiv:1701.06279.*

Nguyen, N. G., Tran, V. A., Ngo, D. L., Phan, D., Lumbanraja, F. R., Faisal, M. R., Abapihi, B., Kubo, M., and Satou, K. (2016). DNA sequence classification by convolutional neural network. *Journal of Biomedical Science and Engineering*, 9(05):280.

Olive, D. M. and Bean, P. (1999). Principles and applications of methods for dna-based typing of microbial organisms. *Journal of clinical microbiology*, 37(6):1661–1669.

Pais, F. S.-M., de Cássia Ruy, P., Oliveira, G., and Coimbra, R. S. (2014). Assessing the efficiency of multiple sequence alignment programs. *Algorithms for Molecular Biology*, 9(1):4.

Park, S., Min, S., Choi, H.-S., and Yoon, S. (2017). Deep Recurrent Neural Network-Based Identification of Precursor microRNAs. In *Advances in Neural Information Processing Systems*, pages 2895–2904.

Pastur-Romay, L. A., Cedrón, F., Pazos, A., and Porto-Pazos, A. B. (2016). Deep artificial neural networks and neuromorphic chips for big data analysis: pharmaceutical and bioinformatics applications. *International Journal of Molecular Sciences*, 17(8):1313.

Pearson, W. R. (2013). Selecting the right similarity-scoring matrix. *Current protocols in bioinformatics*, pages 3–5.

Pevsner, J. (2015). *Bioinformatics and functional genomics.* John Wiley & Sons, Hoboken, NJ, USA.

Rampasek, L. and Goldenberg, A. (2016). TensorFlow: Biology's Gateway to Deep Learning? *Cell systems*, 2(1):12–14.

Rizzo, R., Fiannaca, A., La Rosa, M., and Urso, A. (2015). A Deep Learning Approach to DNA Sequence Classification. In *International Meeting on Computational Intelligence Methods for Bioinformatics and Biostatistics*, pages 129–140. Springer.

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., and Blunsom, P. (2015). Reasoning about entailment with neural attention. *arXiv preprint arXiv:1509.06664.*

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Scholz, M. B., Lo, C.-C., and Chain, P. S. (2012). Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis. *Current opinion in biotechnology*, 23(1):9–15.

Singhal, N., Kumar, M., Kanaujia, P. K., and Virdi, J. S. (2015). [malditof mass spectrometry: an emerging technology for microbial identification and diagnosis]. *Frontiers in microbiology*, 6.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Srivastava, S. (2016). *Genetics of Bacteria.* Springer.

Stollenga, M. F., Byeon, W., Liwicki, M., and Schmidhuber, J. (2015). Parallel Multi-Dimensional LSTM, With Application to Fast Biomedical Volumetric Image Segmentation. *CoRR*, abs/1506.07452.

Tacconelli, E. and Magrini, N. (2017). Global priority list of antibiotic-resistant bacteria to guide research, discovery, and development of new antibiotics. http://www.who.int/medicines/publications/WHO-PPL-Short_Summary_25Feb-ET_NM_WHO.pdf.

Tan, P.-N., Steinbach, M., Karpatne, A., and Kumar, V. (2018). *Introduction to Data Mining.* Pearson Education, London,UK, second edition.

Varghese, N. J., Mukherjee, S., Ivanova, N., Konstantinidis, K. T., Mavrommatis, K., Kyrpides, N. C., and Pati, A. (2015). Microbial species delineation using whole genome sequences. *Nucleic acids research*, page gkv657.

Webb, S. (2018). Deep learning for biology. *Nature Technology Features*, 554:555–557.

Williams, A. (2016). *The docker & container ecosystem*, volume 1. The New Stack.

Woese, C. R., Kandler, O., and Wheelis, M. L. (1990). Towards a natural system of organisms: proposal for the domains archaea, bacteria, and eucarya. *Proceedings of the National Academy of Sciences*, 87(12):4576–4579.

Yuan, Y., Shi, Y., Li, C., Kim, J., Cai, W., Han, Z., and Feng, D. D. (2016). DeepGene: an advanced cancer type classifier based on deep learning and somatic point mutations. *BMC Bioinformatics*, 17(17):243.

Zankari, E., Hasman, H., Cosentino, S., Vestergaard, M., Rasmussen, S., Lund, O., Aarestrup, F. M., and Larsen, M. V. (2012). Identification of acquired antimicrobial resistance genes. *Journal of antimicrobial chemotherapy*, 67(11).

Zhang, K., Alqahtani, S., and Demirbas, M. (2017). A comparison of distributed machine learning platforms. In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*, pages 1–9. IEEE.