THESIS

READ ALIGNMENT USING DEEP NEURAL NETWORKS

Submitted by

Akash Shrestha

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

ABSTRACT

READ ALIGNMENT USING DEEP NEURAL NETWORKS

Read alignment is the process of mapping short DNA sequences into the reference genome. With the advent of consecutively evolving "next generation" sequencing technologies, the need for sequence alignment tools appeared. Many scientific communities and the companies marketing the sequencing technologies developed a whole spectrum of read aligners/mappers for different error profiles and read length characteristics. Among the most recent successfully marketed sequencing technologies are Oxford Nanopore and PacBio SMRT sequencing, which are considered top players because of their extremely long reads and low cost. However, the reads may contain error up to 20% that are not generally uniformly distributed. To deal with that level of error rate and read length, proximity preserving hashing techniques, such as Minhash and Minimizers, were utilized to quickly map a read to the target region of the reference sequence. Subsequently, a variant of global or local alignment dynamic programming is then used to give the final alignment.

In this research work, we train a Deep Neural Network (DNN) to yield a hashing scheme for the highly erroneous long reads, which is deemed superior to Minhash for mapping the reads. We implemented that idea to build a read alignment tool: DNNAligner. We evaluated the performance of our aligner against the popular read aligners in the bioinformatics community currently — `minimap2`, `bwa-mem` and `graphmap`. Our results show that the performance of DNNAligner is comparable to other tools without any code optimization or integration of other advanced features. Moreover, DNN exhibits superior performance in comparison with Minhash on neighborhood classification.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my advisor, Dr. Hamidreza Chitsaz, for his patience, constant support, and encouragement for the accomplishment of this research. I have learned a great deal from his supervision and I will always be grateful to his immense knowledge in Bioinformatics.

I would like to extend a special thanks to my thesis committee members, Dr. Asa Ben-Hur, and Dr. Zaid Abdo, for their enthusiastic encouragement and useful critiques of this research work. I would also like to express my sincere thanks to my colleague, Mr. Ali Ebrahimpour, for his constant support and for compensating the shortcomings in me.

I would like to thank all my friends in CSU for making this a wonderful experience and being a precious part of my life. To name a few, my special thanks to Sujana Timalsina, Rejina Basnet, Swapnil Sneham and Alex Fout. Lastly, my acknowledgment would be incomplete without thanking my family. Thank you for believing in me and constantly supporting me in every possible ways to keep me strong.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Deoxyribonucleic acid (DNA) is the blueprint of life. Understanding genome, which is encoded by the DNA, has leveraged scientific communities to research about complex biological processes of living organisms. Genome sequencing was made feasible with the development of Sanger sequencing [1] in the late 1970s, after which a lot of potential research areas has emerged. Some well-known research areas include evolutionary biology, metagenomics, personalized medicines, forensics and more. Genomic sequences have provided a tremendous understanding of life science, however, the picture of these processes are still very far from being complete.

DNA is a molecule made up of Adenine(A), Guanine(G), Thymine(T) and Cytosine(C), where each of these chemical compounds (nucleotides) are made up of a nitrogenous base, a five-carbon sugar (ribose) and a phosphate group [2]. Genome sequencing is a process of finding the nucleotides in the exact order that make up an organism's DNA. The genome sizes vary significantly between species. The bacterium Escherichia coli K-12 [3] has a genome size of 4 million base pairs, whereas human beings have a genome size of 3 billion base pairs [4]. Current sequencing technologies cannot read the DNA sequence from start to the end, all at once, in a single run. Instead, the DNA is fragmented into many shorter pieces and each fragment is sequenced individually. Later, to connect the pieces together, a computational process called genome assembly is carried out for obtaining a complete genome. Reference genome is a genomic sequence of an organism published by scientists to serve as a standard representative of the organism's genome.

Sequence alignment is a process of mapping shorter DNA sequences into the reference genome. Many important projects in genomics are dependent on sequence alignment. Genome variant discovery [5], quantitative analysis of transcriptome (RNA-seq), identification of protein binding sites (ChIP-seq) [6], genome assembly [7] and study of genome-wide methylation pattern [8] are some of the applications of sequence alignment. The alignment process is inherently hard as it involves searching for small fragments in a relatively large sequence. Further, due to the imperfect nature of sequencing technologies, the alignment process often involves searching for inexact fragments, adding in more complexity. Traditionally, this problem was handled by allowing a few mismatches while searching. For example, using gapped seed search if searching in hash tables or allowing few mismatches while traversing a suffix trie. With the recent advent of third-generation sequencing technologies, the reads of longer lengths are obtained in comparatively cheaper cost, which has been a blessing for the bioinformatics community and has been widely accepted for future research. But the longer read length comes with a price. The reads obtained from third generation technologies are much error-prone (up to $\sim 30\%$) and the error profile is not uniform. The nature of these reads make the traditional approach unsuitable for long-reads alignment.

This thesis presents an approach using Deep Neural Network and Minhash for sequence alignment. The method is inspired by the successful application of Neural Network in pattern recognition from the data. In this project, we want to see if Neural Network can learn patterns from the genomic sequence for alignment in presence of high error rate. On the other hand, Minhash is a probabilistic dimensionality reduction technique, which has been used multiple times in the literature of long reads alignment, correction, and genome assembly. Minhash captures the fingerprints from the document, which provides an approximation of Jaccard similarity between them. This thesis presents a sequence alignment tool - DNNAligner, which uses Neural Network to find the most probable locality for a given read and uses Minhash for mapping it to approximate neighborhood. The results show that the performance of our approach is comparable to other popular tools, suggesting that Neural Network might be one of the areas to explore for development of future tools in the sequence alignment.

The rest of the thesis is organized as follows. The remainder of the introduction presents a discussion about sequencing technologies. In Chapter 2, we discuss some of the related works accomplished for sequence alignment problem. We discuss the approaches those tools have used, and discuss how the subsequent development of other tools tackled the problem faced by the former tools. In Chapter 3, we discuss the sequence alignment problem from a high-level view and discuss inputs and outputs expected by a generic sequence alignment tool. Later we provide a high-level view of our sequence alignment tool. In Chapter 4, we discuss Neural Network architectures, data preparation, and training, and contrast two different architectures we tested. In Chapter 5, we discuss Minhash and how we use it in our aligner. In Chapter 6, we discuss the local alignment problem and discuss the solution based on dynamic programming. In Chapter 7, we discuss the results and evaluation of our sequence alignment tool.

1.1 Sequencing Technologies

The first sequencing technique emerged in the late 1970s and the technology has been evolving ever since, increasing the throughput and reducing the cost of sequencing. The cost of sequencing a human genome was in the range of billions when it was first sequenced. Later on, with technological advancement, the average cost fell to around \$1000 [9].

1.1.1 First Generation Sequencing

The first DNA sequencing method, Sanger sequencing, was published in 1977 [1]. In this method, DNA is fragmented and a complementary strand is artificially synthesized in test tubes using dNTPs (deoxynucleotides). As the DNA is being synthesized, the complementary strands are terminated with ddNTPs (dideoxynucleotide) at random positions resulting in strands of uneven length. The strands are passed through capillary gel electrophoresis, where shorter strands move faster than longer strands. The ddNTPs are fluorescently labeled allowing the nucleotides to be read from the gel, which provides with the sequence of read associated with the fragment. Sanger sequencing is also known as Chain Termination method.

Sanger sequencing provides long and highly accurate reads. The reads are typically ~1000 base pairs long and per-base accuracy is 99.99%. This technology suffers from very low throughput resulting in a very high cost of sequencing (\$0.50 per kilobase) [10]. The errors are dominated by substitutions and are more likely towards the end of the read.

1.1.2 Next Generation sequencing

Next Generation sequencing technologies (NGS), also known as the second generation, came into existence around 2005. These technologies break the limitations of previous technologies providing high throughput (millions of sequencing reads in parallel), low sequencing cost and direct inference of bases without requiring electrophoresis. Based on their sequencing approach, NGS technologies are further divided into Pyrosequencing, Sequencing by synthesis, Sequencing by ligation and Ion semiconductor sequencing. Some of those approaches involve a synthesis of the complementary strand of DNA and reads are produced by sensing the nucleotides consumed on the process.

Roche/454 sequencers use pyrosequencing, which works by releasing a pyrophosphate while synthesizing. The released pyrophosphate undergoes chemical reaction resulting in an emission of light, which is sensed to determine the corresponding nucleotide. Pyrosequencing generates a read of length almost close to Sanger sequencing length but it suffers from insertion/deletion errors due to the presence of homopolymers in a genome [10]. Ion semiconductor sequencing is similar to the Pyrosequencing, except it produces hydrogen ion instead of pyrophosphate.

Illumina sequencers use sequencing by synthesis. In this process, PCR bridge amplification creates several identical copies of each sequence, which are organized into clusters. A synthesis process is carried out on those clusters. At each step, a terminated nucleotide is attached to form a complementary DNA strand. The attachment emits light and is sensed. The synthesis process continues with the removal of a terminator, fluorescent label and washing away entire solution for a fresh new cycle [11]. Illumina produces reads of about 50-300 bp length, and it majorly suffers from substitution error towards the end of the read.

1.1.3 Third Generation sequencing

The third generation sequencing technologies came into existence around 2010. These technologies emphasize single molecule sequencing, which is sequencing from a single DNA molecule without the amplification step. The two main technologies that fall under this generation are PacBio

SMRT (Single-Molecule Real-Time sequencing) and Oxford NanoPore Technology (ONT). PacBio SMRT sequences a strand by sensing the fluorescent signal emitted due to attachment of a nucleotide while synthesizing. However, it does that in real-time without the need for amplification in a microfabricated structure called ZeroMode Waveguide (ZMW). ZMWs are nanometer-sized wells whose bottom is illuminated with a light source. Due to the fact that the diameters of those wells are smaller than the wavelength of light, the intensity of light gradually decreases towards the top, hence reducing the interference from the top. The wells contain DNA polymerase attached to the bottom where strand extension occurs and the emitted signals are detected on the fly. Similarly, ONT, on the other hand, uses nanopores to isolate and detect the bases. Only one DNA molecule can pass through a nanopore and it detects the electric current fluctuation as the molecule passes through it. The nucleotides are inferred from the recorded electric fluctuation. The read length of SMRT sequences is on average ~10kbp long with an error rate around 13% [12]. The individual read can be up to 60kbp in length [11]. ONT sequencers have a high error rate of ~12% and each read might be as long as 150 kbp [11]. Both of those devices are still under active development.

Chapter 2

Literature Review

Since the development of Sanger sequencing technology [1], the sequence alignment problem has been researched extensively. In bioinformatics, two sequences are compared to each other based on the edit operations (insertion, deletion, and substitution) required to transform one sequence to another, also known as edit distance. The sequences with less edit distance are more similar to each other. Further, when a substring of a target string is matched with the query string, then it is termed as a local alignment. The sequence alignment problem is essentially a local alignment between read sequence and the reference genome. The region of the genome, where a read is matched with an edit distance below some predefined threshold, is the region where it is mapped. The local alignment is guaranteed to find all the regions above some threshold and therefore it is considered as a full sensitivity search [13]. The local alignment between two sequences can be computed using dynamic programming (DP) algorithms —Smith-Waterman algorithm [14] and Affine-gap penalty algorithm [15]. SSearch [16] is a sequence alignment tool, which is based on Smith-Waterman local alignment. Due to the enormous length of the reference genome, dynamic programming is infeasible for sequence alignment as the computational complexity of local alignment is proportional to the product of the length of target and query sequences. Furthermore, the local alignment performs an excessive amount of unnecessary computation trying to align a significantly shorter query to an enormously long reference genome.

Altschul et al. [17], in 1990, published a tool called BLAST, that uses a hash-table based seed-and-extend approach to search for potential matching regions in the genome. The fixed-sized k-mers, called seeds, are searched in the hash-table for exact matches to gather all possible hits. The seeds falling close to each other in the reference genome are clustered together and ranked for the further extension using dynamic programming (DP) algorithm to find the true alignment of the query sequence. One remarkable advantage of this approach is that the search is focused on a region where the query sequence is more likely to be aligned. This approach cuts off the expensive DP algorithm in the entire reference genome to a relatively shorter subsequence of the genome. Even though this approach incorporates building a hash-table beforehand and uses extra space for maintaining an in-memory hash-table throughout the search, it drastically improves the performance, making sequence alignment practically feasible. BLAST has been recognized as a universal tool for rapid sequence search in online/offline databases presently. BLAT [18] and FASTA [19] are other popular tools that follow similar paradigm. The performance of these tools depends on the size of exact matching seeds. For a smaller seed size, there will be an enormous number of hits leading to false alignments, which essentially increases the time required for alignment. The use of longer seed size will increase the specificity of search and keep the computation focused in the region of more likelihood, but in contrast, it will reduce the sensitivity of search as there will be fewer hits. Because of those reasons, this approach fails to achieve full sensitivity as obtained by local alignment, despite speeding up the alignment process.

The reason behind failure in achieving full sensitivity is due to the erroneous nature of sequenced reads and inherent genomic variation. Ma et al. [20] discussed the use of inexact k-mers for seeding. This idea involves searching for the seed match using k-mer template. A template is a fixed length sequence of zeros and ones (Figure 2.1). Based on the template, two k-mers are considered match if they have same nucleotides in the position identified by the corresponding position of ones in the template, irrespective of nucleotides at the corresponding position of zeros. This type of seeding approach allowing internal mismatches is termed as spaced seeding. Ma et al. [20] developed a sequence alignment tool called PatternHunter, through which they illustrated that the use of spaced seeds drastically improves sensitivity of the search. Sensitivity comes from the fact that spaced seeds are more likely to discover homologies and are more tolerant with sequencing errors compared to exact matching seeds. They also discussed that the use of multiple spaced seeds will increase the sensitivity at low cost. The core idea behind using multiple spaced seeds in searching is that the hit missed by one template could be captured by others. This process involves building a hash-table for each seed which requires additional memory for each addition of a template. Hence, the challenge behind using multiple spaced seed in searching is the necessity of optimally designed templates. In PatternHunter II [21], Li et al. implemented the idea of multiple spaced seeds by proposing a way to design a near optimal spaced seeds. They provided a dynamic programming algorithm to compute hit probability of k seeds which can be used to compute an optimal set of spaced seeds for a given sensitivity (hit probability). The dynamic programming approach is costly and time-consuming. Hence, they proposed an alternate greedy approach to efficiently compute near-optimal templates on top of few seeds obtained by dynamic programming. Further, Kucherov et al. [22] provided a way to minimize the number of multiple spaced seeds maintaining a level of sensitivity. Similarly, Lin et al. [23] published a tool called ZOOM, through which they presented an optimal way to design templates for a given query sequence length to achieve some predefined sensitivity. They illustrated that their program ZOOM can discover all two-mismatches hits from 36bp read by using 5 spaced seed templates of weight 16.

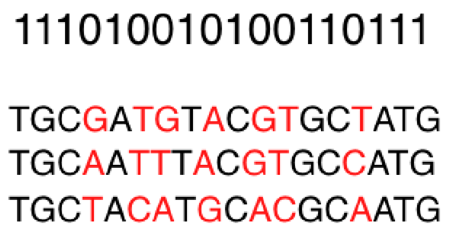


Figure 2.1: k-mer matching with spaced seed templates `111010010100110111` with weight 11 and length 18. The 18-mers shown are considered matches even though exact matching algorithm would consider them different.

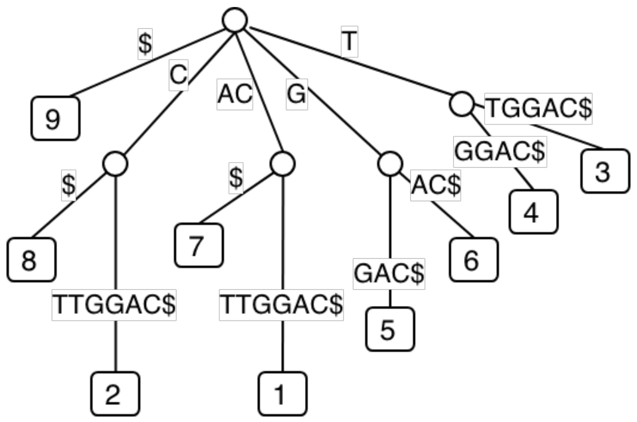
Unlike the approaches discussed so far, there is another class of sequence alignment tools that are based on maximal matches instead of fixed sized seed matches. These aligners are based on string matching using data-structures like Suffix trees, Suffix arrays and FM-index. A suffix tree is a tree data structure that stores all the suffixes of a string as a path from the root to the leaf as shown in Figure 2.2c. The suffix trees can be built efficiently in linear time proportional to the length of the genome. Finding exact matches in the suffix tree is straight forward, which can be achieved by following the path from the root until the query is entirely consumed or tree runs

1 ACTTGGAC$
2 CTTGGAC$
3 TTGGAC$
4 TGGAC$
5 GGAC$
6 GAC$
7 AC$
8 C$
9 $

(a) List of all suffixes with corresponding start coordinates

9 $
7 AC$
1 ACTTGGAC$
8 C$
2 CTTGGAC$
6 GAC$
5 GGAC$
4 TGGAC$
3 TTGGAC$

(b) Alphabetically sorted suffix array



(c) Suffix tree with leaf node representing start coordinates

Figure 2.2: Representation of the string `ACTTGGAC` in different forms based on its suffix.

out of the path. The traversal of the tree by following the characters in the query string returns the coordinates where the query can be mapped in the target string. A single traversal of the suffix tree computes multiple identical regions where query occurs in the target string. This is a remarkable advantage over fixed-sized seeding, where alignment needs to be performed for each cluster of hits. The mismatches can be allowed in the search by allowing traversal to visit nodes even when the character in the query string does not match corresponding edges. This opens up the possibility for graph traversal algorithms like A^* in the sequence alignment. In 2003, Meek et al. published a tool called OASIS [24], which is built on this principle. OASIS stores the entire target sequence as a suffix tree and alignment is performed by traversal of the tree. The traversal is guided by dynamic programming along with best-first(A^*) search. As the partial alignment scores are computed, OASIS computes the upper bound of the cost of matching the rest of the query. The nodes are explored in the order of possibility of achieving maximum alignment score. Similarly, MUMmer [25, 26] is a whole genome comparison tool that uses a suffix tree. MUMmer computes alignment between two whole genomes that are very closely related and share sequence homology. MUMmer first computes the Maximal Unique Match (MUM) between two genomes. A MUM is defined as a subsequence that occurs once in both genomes and is not contained by any other longer sequences. MUMs are computed using suffix trees and are sorted based on their position in genomes. The longest increasing subsequence (LIS) of matches are connected by closing the gaps using dynamic programming to compute the one-to-one alignment of two genomes.

The storage requirement of the suffix tree of a genome is in the order of its length. To tackle this problem, Abouelhoda et al. [27] proposed an alternative representation called enhanced suffix array. This approach involves representation of genome using suffix array and lcp-table for storing the longest common prefix. They showed that any algorithm which requires top-down traversal of suffix tree can be replaced with an equivalent algorithm in enhanced suffix array. Later, they published a tool, VMatch [28], which uses an enhanced suffix array. Through the VMatch, they showed that an enhanced suffix array provides more space efficiency and faster running time. Segemehl [29] is another sequence alignment tool which uses enhanced suffix arrays.

Equivalently, Burrows-Wheeler Transformation (BWT) [30] and FM-index [31, 32] are approaches for searching maximal matching strings. BWT is a special permutation of original string such that resulting transformation may contain runs of repeated characters. This allows compression of the original string. The original string can be computed using the permuted string alone without any additional information, in a cost-effective manner. For computing the BWT, the text is cyclically rotated to collect all cyclic rotations of the text, followed by lexicographical sorting of those cyclic rotations. The BWT is taken as the sequence of last characters from those sorted rotations as shown in Figure 2.3. FM-index [31, 32] is an indexing technique based on BWT that allows to search for subsequences, similar to traversal in the suffix tree. FM-index contains tables which indirectly represent the columns F and L as in Figure 2.3. One important property of BWT is that the i^{th} occurrence of a character in F is also i^{th} occurrence in L. Each occurrence of a unique character in the target sequence falls in a continuous interval in the F table, as they are alphabetically sorted. While searching a subsequence using FM-index, the search begins by searching from the shortest suffix of the query string and successively extending the suffix. For example, The last character of the query sequence (a character in suffix) can be located in F table to obtain an interval in the target, where that character occurs. Within that interval, L table gives the next character right before the current suffix. This allows to gradually narrow down the interval. The $L - to - F$ mapping table helps to map i^{th} occurrence of a character in L to i^{th} occurrence in F. The exact matching search proceeds this way by gradually extending the suffix and it finally returns an interval, giving occurrences of the query string in the target. The interval might be empty before all the characters in the query sequence are consumed, which signifies that the query does not occur in the target. Bowtie [33] is a sequence alignment tool based on

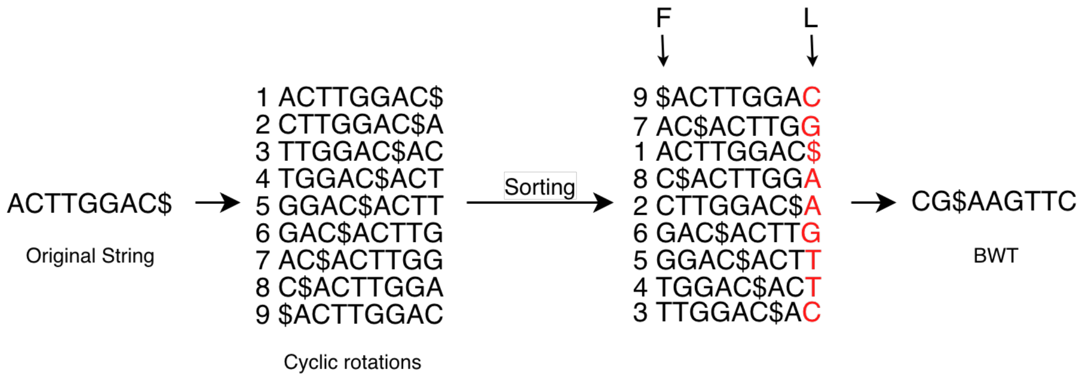


Figure 2.3: Burrows-Wheeler Transform of string `ACTTGGAC`

searching using BWT and FM-index. The search proceeds as described above, by searching for exact matches until the interval is empty. After that, the search process replaces a character, that is already matched in the suffix, with a different character, which signifies the allowance of mismatch in the search. The character to be replaced is decided based on the quality score information provided by the sequencing machine. The exact matching continues after replacing the character, which introduces numerous branches in the search. To deal with those branches, Bowtie introduces the double indexing technique. It creates an FM-index for the target string, called a forward index, for searching as discussed above. The other index is called a mirror index, which is FM-index of the reversed sequence of the target string (not DNA complement). Using those two indices, the exact matching search is performed from both ends of the query string. Similarly, BWA [34], BWT-SW [35] and SOAP2 [36] are other sequence alignment tools based on BWT and FM-index. One remarkable advantage achieved through BWT and FM-index is the data compression. As mentioned in SOAP2 [36], the sequence alignment of human genome consumed only 5.4 GB of memory compared to their previous algorithm (based on hash-table) in SOAP [37], which required 14.7 GB. Further, the FM-index of human genome alone required only 1 GB of main memory, as reported on BWT-SW [35].

The sequence alignment problem has been addressed by many researchers for a long time. Beyond all the classes of alignment tools discussed above, there are other varieties of sequence alignment tools. Such as: Slider [38]/Slider II [39] based on merge sorting and SHRiMP [40] based on q-gram filtering [41]. All these approaches discussed so far, are for solving sequence alignment problem of reads generated by second (next) generation sequencing technologies (NGS). The reads from NGS are characterized by their short length and significantly less error rate. With the advent of new technologies, currently, the third generation of sequencing technologies are becoming more popular. These technologies are well recognized for their long length. As discussed in Chapter 1, many applications in bioinformatics are benefited by the use of long reads. Due to their erroneous nature, they possess a different nature of problem. The sequence alignment tools developed for NGS reads cannot handle, or are very inefficient for, aligning the long reads. For example, the best-first traversal of suffix tree will need to incorporate more mismatched nodes, which means more memory and time for finding true alignment. Similarly, Bowtie [33] will require to backtrack a massive number of possible exact matches in FM-index, which is probably not feasible. Although the manufacturers of these technologies are gradually making them better with fewer errors, there has been some research to solve the long reads alignment problem algorithmically. This class of sequence alignment tools is broadly classified into two categories. Some of these tools are adapted from their NGS counterpart with few heuristics well suited for long reads alignment. Other tools are genuinely built for long reads.

Basic local alignment with successive refinement (BLASR) [42] is a sequence alignment tool which works by successive refinement of seeds. BLASR begins by finding all the exact matches between the genome and reads with a length of at least some predefined k. The exact matches are computed by computing the Longest Common Prefix(LCP) using BWT/FM-index or Suffix Array between genome and reads. The LCPs of length below k are discarded while those LCPs longer than k, called as anchors, are clustered for further refining. For clustering, the anchors are sorted based on their position in the genome and a maximal subset of anchors spanning over the region, that is almost as long as the query string, are computed using global chaining [43]. The clusters are ranked based on the constituent anchors. The anchors (LCPs) which occur more often in the genome is assigned a lesser score such that they rank lower. The top clusters are further considered for computing alignment. First, a sparse dynamic programming [44] algorithm re-aligns and re-scores each of the cluster and the top clusters from those are taken for base-to-base alignment using banded DP. The banded DP is guided by the layout of anchors computed by Sparse DP. BLASR employs an exact matching technique similar to NGS read alignment tools but extends the idea using heuristics like global chaining, ranking, and sparse dynamic programming to make it more suitable for long reads.

DALIGNER [45] is a sequence alignment tool designed for aligning long reads. It uses a similar filtration approach as BLASR. It finds exact matching k-mer seeds between query and target by sorting the k-mers and merging the sorted list of k-mers. For sorting, it uses highly optimized threaded radix sort which is much cache coherent and more sophisticated than BWT-FM or Suffix Arrays in BLASR. Similarly, GraphMap [46] is another sequence alignment tool specifically designed for nanopore sequencing technology. It follows a seed-and-extend paradigm with successive refinement heuristic like BLASR [42]. For seeding, it uses gapped seeds like in PatternHunter [20, 21]. The seeds are bundled together into anchors by a graph-based approach incorporating mismatches. The anchors are extended through chaining using k-mer version of

Longest Common Subsequence (LCS). The LCS is further refined with an L1 linear regression such that the anchors are arranged into a diagonal. The final alignment is constructed through dynamic programming between the cluster endpoints.

MHAP [7] is a sequence alignment tool based on Minhash [47], which allows comparing two documents by hashing them. The documents are hashed by multiple hash functions and the minimum hash values obtained for each function forms the representation of the document in reduced dimension. Broader(1997) [47] provided a proof showing that the similarity between documents can be estimated using the minimum hash values alone. MHAP uses Minhash to compare the query and target strings by using hash-tables based approach of searching. Due to the reduced dimensional representation of the documents using Minhash, the size of the index is drastically smaller than the hash-tables used in BLAST. Further, the internal parameters of Minhash can be tuned to increase/decrease the sensitivity of the search. Minimap [48]/Minimap2 [49] uses the hashing concept similar to MHAP, but it uses the minimizers [50, 51]. Given a fixed-length sequence, it forms an overlapping sliding window of size w. The minimizers are the list of minimum k-mers in those successive sliding windows. Minimap2 uses the concept of sorting the seeds similar to DALIGNER, successive refinement similar to GraphMap and performs global chaining similar to BLASR. For base-to-base alignment, it performs dynamic programming using Z-drop heuristic, which is similar to X-drop in BLAST [17] except that it does not drop alignment due to the presence of long gaps.

In the following chapter, we provide the formulation of our sequence alignment tool, which is based on Deep Neural Network and Minhash. We discuss that the ability of neural networks in pattern discovery can be applied to genomics for aligning the query sequences to the target sequence.

Chapter 3

Formulation of genome alignment

The inputs that any generic sequence alignment tool requires is reference genome and the set of sequenced reads. Based on the input, the alignment tool should report the starting coordinates and the strand orientation for each read that is mapped or report as unmapped. Additionally, mapping quality (confidence about mapped coordinate) and alignment report (how well does the read aligns with reference) might also be reported as they are used by downstream applications. In our approach, the provided reference genome is used to train a Deep Neural Network (DNN) and to build the hash-table to search using Minhash. Likewise, the input reads are processed through the pipeline as shown in Figure 3.1 to report the final alignment.

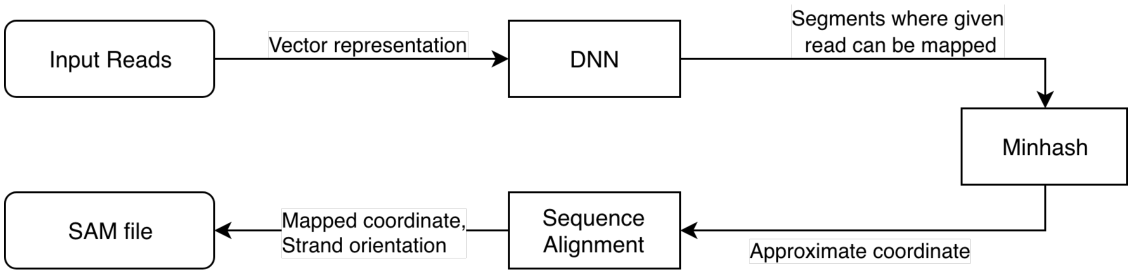


Figure 3.1: High-level view of components in DNNAligner showing the pipeline of alignment

First of all, each chromosome in the provided reference genome is divided into discrete localities, called segments. DNN is trained to classify reads to a set of segments where they belong. For each input read, DNN provides the segments which have a high score of containing the read or its correct version. It allows the searching process to be concentrated in a specific segment, which is more efficient than searching in the entire genome. Next, the reference genome along with the segment information is used for building an inverted index. The index helps to search for the approximate coordinate of each input read within a segment. Minhash is used for this purpose, which is a dimensionality reduction technique and it provides an approximation of Jaccard similarity measure for computing similarity between two documents. The inverted index is searched for matches corresponding to both positive and negative orientation of the input reads. Later to find the true alignment, one-to-one sequence alignment is carried out using dynamic programming (DP) algorithm, which gives the starting coordinate, mapping quality, and alignment report. The alignment and the related information are reported in a Sequence Alignment Map (SAM) file.

3.1 Vector representation of DNA sequence

A genomic sequence, S, is a string of arbitrary length where $S \in \Sigma^*$ and $\Sigma = \{A, C, G, T\}$. A k-mer on string S, is represented as S_i^k which shows a substring of length k starting at index i. For a given length k, the set of all possible k-mers contains 4^k elements. For each string S, its k-mer representation is a binary vector V of size 4^k where the i^{th} element of vector V corresponds to the i^{th} k-mer in the alphabetically sorted array of all k-mers in Σ^* which we represent as $\Sigma_{srt}^*[i]$. So, $V_i = 1$ if and only if $S_j^k = \Sigma_{srt}^*[i]$ for some j $(0 \leq j < |S| - k + 1)$ and $V_i = 0$ otherwise. Further, each alphabet in DNA can be given a numeric value, for example, $f(\Sigma)$ = 0, 1, 2, 3 for A, C, G, T respectively. Then for a given S_j^k, the index i in the vector V can be computed as $i = \sum_{p=0}^{k} f(S_j^k[p]).4^p$. The vector representation can be computed in a single pass through the string S, as the k-mers are identified, as shown in Figure 3.2.

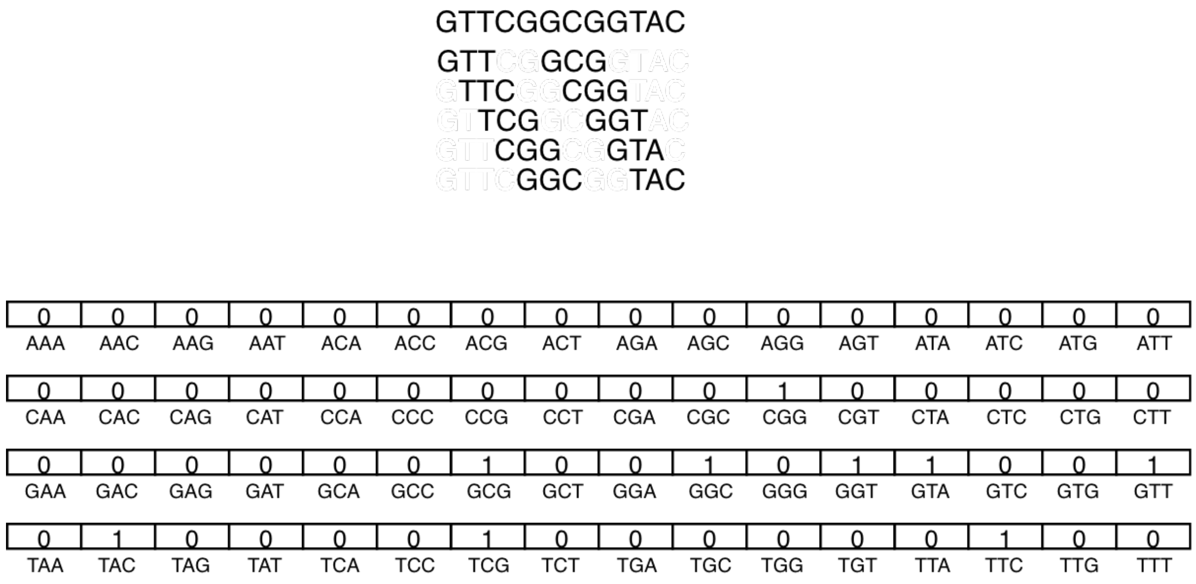

GTTCGGCGGTAC
GTT GCG
TTC CGG
TCG GGT
CGG GTA
GGC TAC
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT ATA ATC ATG ATT
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
CAA CAC CAG CAT CCA CCC CCG CCT CGA CGC CGG CGT CTA CTC CTG CTT
0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 1
GAA GAC GAG GAT GCA GCC GCG GCT GGA GGC GGG GGT GTA GTC GTG GTT
0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0
TAA TAC TAG TAT TCA TCC TCG TCT TGA TGC TGG TGT TTA TTC TTG TTT

Figure 3.2: Vector representation of a DNA sequence `GTTCGGCGGTAC` with k=3

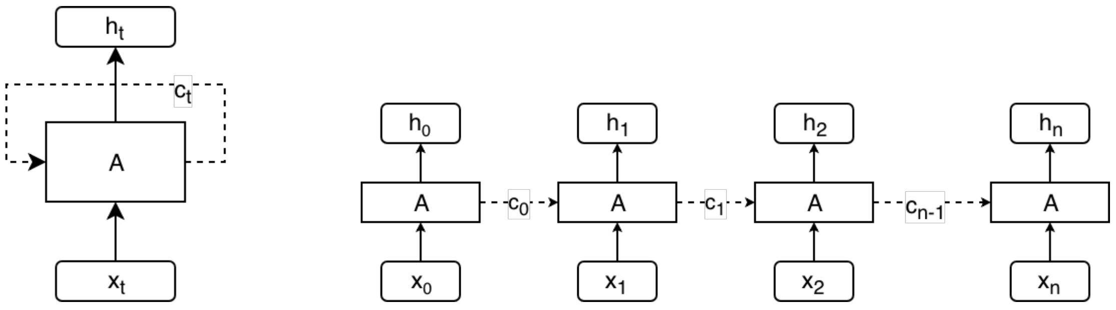
Chapter 4

Neural Network

Neural Network is a class of algorithm in machine learning, which learns a representative function based on the provided data. A Neural Network is built up of units called neurons. Each neuron takes some input and produces an output called activation. The value of the output depends on the inputs and the parameters associated with the neuron. The Neural Network is a collection of neurons connected together such that the activation of one neuron is input to the other neurons. Neural Network is considered a universal function approximator [52].

Feed forward neural network is a class of neural network in which neurons are bundled into layers such that the input to the neurons in layer $n + 1$ is the activations from neurons in layer n. Feed forward architecture is also called as Multilayer Perceptron (MLP). The first layer, called an input layer, is fed with the input data points. The final layer, called an output layer, is the output of the function that the neural network learns. The layers in middle, which do not interact with input/output directly, are called hidden layers and a total number of those layers define the depth of the network. This creates a large mesh of forwardly propagating signals. Mathematically, if x represents an input to a layer in n-dimension, m represents number of unit in that layer, W represents a weight matrix $W \in \rm I\!R^{n \times m}$, b represents a bias vector $b \in \rm I\!R^{m}$ and $\sigma(\cdot)$ represents the activation function, then the activation of that layer $h, h \in \rm I\!R^{m}$, can be defined as $h = \sigma(Wx + b)$.

On the other hand, Recurrent Neural Network (RNN) is a type of neural network architecture that is more suitable for capturing the notion of context from the data. This type of architecture is used mostly for learning the functions which involve time series data. Consider an example of a neural network trying to predict if a sentence has a positive or negative sentiment. The sentences can be modeled as a time series in the sense that one word comes after another in sequence. Consider the sentences, *"I am feeling good"* and *"I am not feeling good"*, which have positive and negative sentiments respectively. Both sentences are exactly the same except for the presence of the word *"not"* in the context. The sentences are represented as binary vectors that tell if the word is present or not. If the MLP is trained with these inputs, it might learn to classify both sentences as having positive sentiment because there were numerous observations which contain *"feeling good"* in positive class than in negative class. MLP disregards the notion of context and fails to learn from this data set. RNN, on the other hand, provides the ability to model context in the neural network. The architecture of RNN is as shown in Figure 4.1a [53]. Let x_t be an input to RNN unit A, at time step t, it learns some internal parameters and outputs h_t and c_t. The output c_t from the unit signifies the context observed so far, for a given sequence of inputs. The context c_t is looped back into the unit and is used as an input along with x_{t+1} for computing h_{t+1}. This way RNN can incorporate the contextual information in learning from data. In real life, the contextual dependency might run over a long span, but in practice, RNN is implemented as shown in Figure 4.1b. The context is propagated only for finite time steps, which simplifies the mathematical computation.



(a) RNN **(b)** RNN unrolled for finite time steps

Figure 4.1: RNN architecture. A is a unit in RNN. x_t is input at time-step t and h_t is output of RNN for input x_t.

The Recurrent Neural Network, as described above, has a limitation in terms of computation. While training the neural network, the parameters are updated by propagating the gradient of the loss function, which is also known as back-propagation. The long-term dependency involves propagating the gradient over numerous time-steps. But, due to the fact that the gradient of the loss function decays exponentially over time steps, the RNN cannot learn long-term dependencies. This problem is referred to as the "Vanishing Gradient" in literature [54]. LSTM (Long Short

Term Memory) networks are a special variety of RNN that allows learning with very long-term dependencies. LSTM units have additional trainable parameters that decide what needs to be forgotten or retained from the context. Further, the RNN unit, as described above, can only model the contextual dependencies from previous time-steps. On the contrary, the bi-directional RNN units can incorporate contextual dependencies from both past and future time-steps. The other popular variety of LSTM is stacked LSTM, where a unit is stacked on top of another unit such that the output of one unit is input to another unit. Stacked LSTM provide a notion of layers, similar to the one in MLP.

All these different varieties of RNN provides the output h_t for some input x_t. The output can be used in multiple ways depending on how we model the problem in terms of the neural network. For example, consider the problem of text generation using RNN. Both the input and output vectors can be modeled in a similar way as described in Section 3.1. Instead of k-mers, the i^{th} element in vectors ($|V| = N$) will be 1 if the input is i^{th} word in dictionary of size N. The output can be directly used to infer the next generated word. For classification problem, like in sentiment analysis, the outputs from all time-steps are combined into one vector and is fed to MLP as input. In this case, the RNN learns the context and produces a transformed representation of input time-steps. MLP takes the transformed representation and learns to classify them to their proper classes. The organization of RNN for classification is as shown in Figure 4.2.

Deep Neural Network (DNN) is used for solving a classification problem in DNNAligner, as discussed in Chapter 3. The reference genome is divided into multiple segments, which are identified by unique segment identifiers and they are treated as different classes in the classification problem. From each of the segments, all the fixed length reads are extracted and the neural network is trained to classify those reads to their corresponding segments. The genome of an organism consists of repeats, hence one read sequence might belong to multiple segments. In other words, one input data point can be assigned multiple class labels. Hence, the DNN is trained to perform a multi-class multi-label classification problem. For solving this classification problem, both MLP and LSTM were tested and the comparative results are discussed in the result section of this chapter.

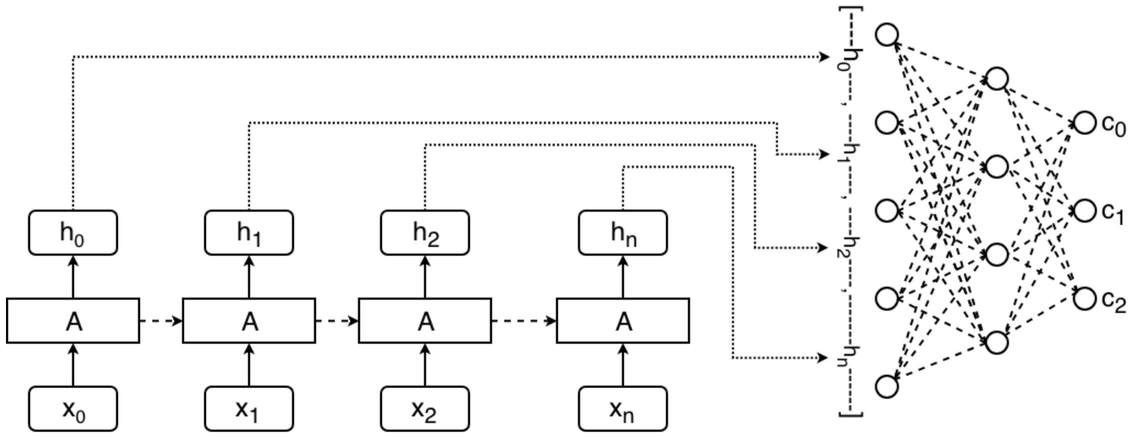


Figure 4.2: Use of RNN for classification. The RNN unit learns a transformed representation from input time-steps. MLP learns to classify the transformed representation into corresponding class labels.

4.1 Data preparation

The data for training the neural network is prepared by processing the reference genome. Let C_L be the length of a chromosome in the provided genome, L be the length of each segment and W be the length of each read sequences to be extracted from the segment, then the chromosome is divided into segments such that the start and end coordinates are given by:

$$start : i \times \left(L + \left\lfloor \frac{W}{2} \right\rfloor\right) \qquad end : start + L + W$$

$$0 \leq i < \left\lceil \frac{C_L - W/2}{L + W/2} \right\rceil$$

For the last segment, when $i = \left\lceil \frac{C_L - W/2}{L+W/2} \right\rceil - 1$, if the segment spreads beyond the last nucleotide, the start point is shifted towards left such that $end - start = L + W$. Each segment is of length $S_L = (L+W)$ except when the chromosome is shorter than $(L+W)$. S_L accommodates the reads falling in the border between two segments by allowing $W/2$ overlap on both ends. We use S_L for the computational purpose, whereas, L is used as a parameter provided while preparing data for training. Although, both of them refers to the length of the segment. From each segment, reads of length W are extracted by taking a sliding window, which slides by some fixed-size $Stride$. The stride is also a parameter we provide while preparing data.

To demonstrate an example of dividing a chromosome into segments, let us assume a chromosome of length, $C_L = 11000$ bases. The chromosome is divided into segments of length, $L = 2000$ and the read sequences of length, $W = 200$, are to be extracted from each segment. The total number of segments can be computed as:

$$\left\lceil \frac{C_L - W/2}{L + W/2} \right\rceil = \left\lceil \frac{11000 - 100}{2000 + 100} \right\rceil = 6$$

For segments, $0 \leq i < 6$, the start and end coordinate pairs are: $(0 - 2200)$, $(2100 - 4300)$, $(4200 - 6400)$, $(6300 - 8500)$, $(8400 - 10600)$, $(10500 - 12700)$. Notice that the last segment spreads beyond the last nucleotide in the chromosome, so the start coordinate is shifted towards the left to get the new coordinate as $(8799 - 10999)$. Also, notice that each segments are of length $S_L = L + W = 2200$ bases.

Next, the segments are introduced with noisy data points to allow the neural network to generalize for a real-world scenario, where the input reads are inaccurate. Each reads in all the segments are mutated with random artificial insertion, deletion and substitution errors. The number of such artificial copies per read (n') and their error rate (e) are both parameters provided while preparing the data. The artificial reads are added back into the pool and their class labels are the same as the labels of corresponding original reads. Additionally, an extra class is created for the reads which do not belong to this reference genome. The reads for this class are populated by randomly sampling all the segments and introducing a lot of mutations to them. We take the rate of mutation for this class to be greater than twice the provided error rate parameter. Finally, there are $N + 1$ classes with original and noisy data points, which is used for training the neural network. To make sure each class has an equal number of reads, the classes having fewer reads are balanced with the addition of more artificial reads.

The extracted read sequences are represented as vectors, as discussed in Section 3.1. For training the MLP, the read sequences ($|S| = W$) are represented as a single vector containing all the k-mers, as described. For training LSTM, the read sequences are represented as time-series of k-mers in the order they occur. Each time-step contains a vector of size 4^k with the corresponding k-mer position represented as 1 and 0 in other positions. The class label (L) is also represented as binary vector of size, $|L| = (N + 1)$. The i^{th} element of vector L is 1 if the read belongs to i^{th} segment and 0 otherwise. Aggregating all the parameters required for data preparation for the given reference genome, they can be listed as: L for length of each segment, W for length of each read window, $Strides$ for sliding window, k for k-mer vector representation, n' for number of artificial copies per read and e for error rate of noisy data.

4.2 Training

The prepared data set from the provided reference genome is split into train and test set in 80-20 proportion. The train-set is used for training a DNN model while test-set is used for testing the trained model. Due to the huge size of entire data set, training the neural network multiple times on the entire training set with different hyper-parameter configuration is very time consuming. Therefore, the training set is randomly sampled to form a relatively small subset of training set. Further, this subset is split into 80-20 proportion, of which the larger one is used for training the network and smaller one is used for validation. Multiple sets of hyper-parameter configurations are tested on this subset followed by validation of the model. The parameter configuration with acceptable performance in validation set is trained with the entire 80% of the training set and tested.

For modeling the multi-class multi-label classification, the Sigmoid function is used as an activation function in the output layer. Sigmoid function can be defined as $\sigma(x) = \frac{1}{1+e^{-x}}$. The value of $\sigma(\cdot)$ falls in the range $[0, 1]$, which provides the score of whether the read belongs to a class. Each i^{th} neuron in the output layers acts as a binary classifier that decides if the read belongs to i^{th} segment out of $(N + 1)$ such binary classifiers. The loss is computed using the cross-entropy loss function. The cross-entropy loss function is popularly used in classification problem as the gradient of this function does not depend on the gradient of the activation function and it depends only on neuron output, target label and neuron input, which prevents slow learning and avoids the vanishing gradient problem that is often faced in deep neural networks. Let y be the predicted label and y' be the true label, then the cross-entropy loss function can be defined as:

$$Cross - Entropy(y, y') = -y' \times log(y) - (1 - y') \times log(1 - y)$$

The neural network is trained using Stochastic Gradient Descent(SGD) approach by minimizing the cross-entropy loss over a mini-batch of training samples. Each mini-batch is composed of randomly chosen samples from the pool of training set. Further, dropout is used for making sure the DNN does not over-fit on the training set. Dropout is a popular methodology for regularization in the neural network, in which the activation of few neurons is forced to be zero. This is equivalent to turning a neuron off, which reduces interdependent learning among neurons. The DNN is implemented, trained and tested using Tensorflow 1.6 [55].

4.3 Results

In this section, we give a brief comparison of LSTM and MLP models by measuring their performance in different network configurations and values of k. The neural network models are compared through recall and precision obtained on the test set. Let n be the total number of labels in each data point, Y be the vector of true labels and Z be the vector of predicted labels, then precision and recall of a model can be defined as:

$$Precision = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Z_i|} \qquad Recall = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Y_i|}$$

The data set is prepared using the E.coli. genome, with the parameters: $L = 5000$, $W = 200$, $n' = 10$, $e = 10\%$ and different values of k. The prepared data sets are used for training multiple neural network configurations. Later, the test set is used to compute precision and recall for each model, which are shown in Table 4.1 and Table 4.2. The models used for training are described below:

- LSTM-Model-1: LSTM cell having hidden state represented as a vector of 20 elements. The k-mers occurring in sequence in the reads of length $W = 200$ are treated as input time-steps and LSTM is unrolled for those time-steps. The hidden states from all time steps are aggregated as a single vector and fed into MLP having hidden layers of $[1000]$ units.

	k	Precision	Recall
Model 1	4	0.7334	0.0990
	5	0.7554	0.1699
	6	0.8253	0.1710
Model 2	4	0.7886	0.1016
	5	0.8253	0.1599
	6	0.8381	0.2389
Model 3	4	0.7747	0.1070
	5	0.8441	0.3049
	6	0.8554	0.3211

Table 4.1: Precision and Recall for different LSTM models with $k = \{4, 5, 6\}$.

	k	Precision	Recall
Model 1	5	0.7112	0.2750
	6	0.7527	0.5078
	7	0.7766	0.6753
Model 2	5	0.8858	0.2720
	6	0.9454	0.6312
	7	0.9809	0.9322
Model 3	5	0.8867	0.3110
	6	0.9223	0.6541
	7	0.9825	0.9378

Table 4.2: Precision and Recall for different MLP models with $k = \{5, 6, 7\}$.

- LSTM-Model-2: Similar to LSTM-Model-1. Two LSTM cells stack on top of each other both having hidden state represented as a vector of 100 elements. The MLP consists of hidden layers with $[3000, 1000]$ units.

- LSTM-Model-3: Single unstacked LSTM unit with hidden states represented as 200 dimensional vector. MLP consists of hidden layers with $[2000, 1000]$ units.

- MLP-Model-1: The MLP consists of one hidden layer with 1000 units. The output layer consists of same number of units as number of classes in the data set.

- MLP-Model-2: Two hidden layers with $[2000, 500]$ units.

- MLP-Model-3: Two hidden layers with $[2000, 1000]$ units.

From the above tables, it can be observed that increasing the size of k-mer, increases the performance of a model. The longer k-mer size decreases the probability of occurrence of a k-mer and makes their occurrence more specific. Hence, the models trained with larger k-mer size learns to distinguish classes with high specificity, which thereby increases the recall. Further, increasing the value of k increases the dimension of vector representation exponentially, making it hard to train the network in commodity machines. From the above tables, it can also be observed that the values of precision and recall for LSTM models are comparatively less than the values for MLP models. For equal k, MLP performs far better than LSTM. For example, both LSTM-Model-1 and MLP-Model-1 have the same number of hidden layers and hidden units. LSTM model has an extra layer of LSTM unit before MLP for learning the context. However, the recall for the MLP model is greater than the LSTM model. The MLP models are simpler than LSTM models and can be easily trained. It should also be noted that the MLP can be trained with higher values of k in commodity machines without running out of memory. The best performance in the E.coli. data set is observed for MLP-Model-3 with $k = 7$.

Chapter 5

Minhash

Minhash is a dimensionality reduction technique. It captures the fingerprints from documents and allows to compare them efficiently with reduced computational cost. Minhash was originally proposed for computing resemblance between textual documents [47]; however, it has been successfully utilized in other domains, such as clustering images [56] and audio [57]. For a genomic sequence alignment problem, the individual reads of DNA need to be mapped to their corresponding locus in the genome. This problem can be modeled as a sequence comparison problem by considering all the fragments of reads starting at different locations in the reference genome as individual sequences. The read to be aligned is compared against all those possible fragments and the fragments with higher similarity are considered for further analysis. The comparison of a read against all the possible fragments is a very tedious process. The vector representation of a read has a size of $|V| = 4^k$ and comparing those vectors one to one can be computationally expensive even for small k. Using the Minhash, each sequence in the genome and the read are represented with a vector of smaller size while retaining enough information to compute their resemblance.

Document similarity is a numeric score computed between two documents to provide a notion of how close they are to each other. Given a universe of documents U, document similarity is defined as $S : U \times U \Rightarrow [0, 1]$. There are different popular metrics in the literature that represents document similarity, such as — Cosine Similarity, Jaccard Similarity. Cosine similarity gives the cosine of an angle between two vectors. The angle between two similar vectors is ≈ 0, which is signified by the cosine value being closer to 1. The angle between two dissimilar vectors is close to orthogonal, which is signified by the cosine value being closer to 0. Similarly, Jaccard similarity is defined for a set and computed as the ratio of size of the intersection of sets to the size of the union of those sets. The documents which are highly similar have a larger intersection size, which makes similarity score closer to 1, whereas, the dissimilar documents have smaller intersection size making the score closer to 0. The formal definition of those metrics are as shown below:

Cosine Similarity:

$$Cosine(A,B) = cos(\theta) = \frac{\mathbf{A}.\mathbf{B}}{\|\mathbf{A}\|\,\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} a_i b_i}{\sqrt{\sum\limits_{i=1}^{n} a_i^2}\sqrt{\sum\limits_{i=1}^{n} b_i^2}}$$

Jaccard Similarity:

$$J(A,B) = \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|} = \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A}| + |\mathbf{B}| - |\mathbf{A} \cap \mathbf{B}|}$$

Given a vector representation of size 4^k or reduced dimensional representation of two sequences, both the above-mentioned metrics can be used to compute the similarity between them. The computation involving Minhash is cheaper because of compact representation in a lower dimension. Further, in the sequence alignment problem, all the sequences having a score above some threshold are of interest. To obtain all of those sequences from the target string, instead of comparing the query sequence with every other subsequences of the target sequence, a hash-table based approach is more suitable. Therefore, the similarity computation using vector representation of size 4^k is similar to the exact matching search using hash-table, as used in BLAST [17], BLAT [18] and FASTA [19].

Minhash uses a probabilistic hashing technique for dimensionality reduction. Probabilistic hashing means that the documents which are closer to each other are hashed into the same bucket with high probability, whereas dissimilar documents are hashed into different buckets. Minhash uses numerous hash functions to hash the input vector. The minimum values of those hash functions are taken as the representation of the document in the reduced dimension. The minimum hash values captured by those hash functions are called as document fingerprints. It can be shown that the Jaccard similarity score computed with those fingerprints is proportional to the score computed using the original 4^k representation. For sequence similarity, the Minhash fingerprints are indexed into a hash-table and searched using the approach similar to exact matching search. The other remarkable advantage of Minhash, after dimensionality reduction, is that the exact matching search technique involves inexactness in itself. Minhash represents the similarity between documents using probability distribution over hash functions. The hash collisions between similar documents are the key to allow inexactness in the search.

For a given vector representation V of a document with size $|V| = n$ where $V_i \in \{0, 1\}$ is the i^{th} element and $0 \leq i < n$, the hash function $h(\cdot)$ maps V_i into an integer in range $[0, n)$. Without loss of generality, it is assumed that there is no collision and each index of the vector is uniquely mapped to the range $[0, n)$ by the hash function. The hash function $h(\cdot)$ actually creates a permutation of the given vector V. The minimum index, $h(i)$, where the value of $V_{h(i)}$ is 1 is considered as the fingerprint captured by $h(\cdot)$ from the given document. Minhash uses N number of these hash functions to capture N fingerprints, which is called a Minhash Sketch. The document is represented using those Minhash Sketches in the reduced dimension. However, the hash functions that are commonly used have collisions and do not map the indices uniquely in the given range. This might result in the selection of incorrect Minhash fingerprint for a given hash function. But, by using numerous hash functions, the fingerprint misrepresented by one of the hash function will be retained by other hash functions.

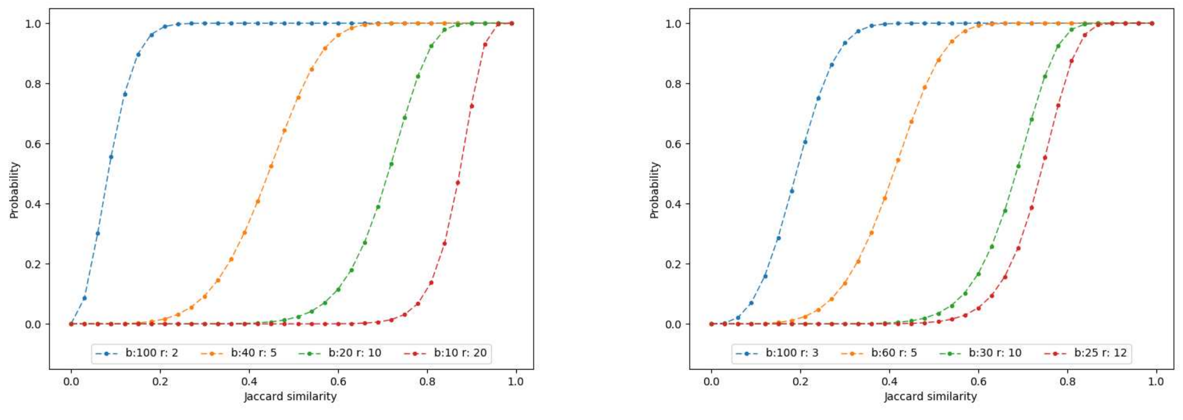
In the real world, implementing the Minhash approach as discussed above is inefficient as it involves permutation of the entire vector, sorting the elements and linearly scanning it afterward to compute the minimum index. This computation needs to be performed for N hash functions. Furthermore, the size of the vector grows exponentially as k increases, making it practically infeasible. Therefore, instead of permuting the entire vector, we only permute the indices of the element which has a value of 1. We apply hash functions $h_1, h_2, ...h_N$ on each of those indices to get N permutations. Also, instead of sorting the entire permuted vector, we keep track of N minimum indices found by each hash function as we go on computing the permutations. Finally, after applying the permutation on all the eligible elements of the vector, we have a list of N minimum indices which are the Minhash Sketch of a document. Our implementation for computing Minhash signature is as shown in Algorithm 1. This implementation is similar to the one in MHAP [7], except that they have used the `XORShift` random number generator as hash function whereas we

```
Function Encode_Kmer($kmer$):
    $sum \leftarrow 0$
    for $i \leftarrow 0$ to $|kmer| - 1$ do
        $sum \leftarrow sum +$ Numeric($kmer[i]$) $\times 4^i$
    end
    return $sum$
end

Function Compute_Minhash($s$, $k$, $Num\_Hash$):
    $hashes \leftarrow \emptyset$
    $best\_min \leftarrow \emptyset$
    for $i \leftarrow 0$ to $|Num\_Hash| - 1$ do
        $best\_min \leftarrow best\_min \cup \{\infty\}$
        $hashes \leftarrow hashes \cup \{\infty\}$
    end

    for $i \leftarrow 0$ to $|s| - k + 1$ do
        $numeric\_kmer \leftarrow$ Encode_Kmer($s[i : i + k]$)
        $seed \leftarrow numeric\_kmer$
        for $j \leftarrow 1$ to $Num\_Hash$ do
            $hash \leftarrow$ rand_r($\&seed$)
            if $hash < best\_min[j]$ then
                $best\_min[j] \leftarrow hash$
                $hash[j] \leftarrow numeric\_kmer$
        end
    end
    return $hashes$
end
```

Algorithm 1: Algorithm for computing Minhash sketches

are using `rand_r()` provided in `stdlib.h`. The pseudo-random number generator is seeded with an index i. The next value it generates is based on the seed and for all the indices, the numbers it generates are almost unique except for few collisions. This is equivalent to applying a hash function to get a new permutation of the vector. Generating N random numbers from a given seed is equivalent to applying N hash functions. Hence, a pseudo-random number generator can be used as multiple hash functions.

Minhash is one of the approach among a family of dimensionality reduction methodologies using hash functions, which are collectively termed as Locality Sensitive Hashing (LSH). In our implementation of Minhash, we used the pseudo-random number generator as the hash function to obtain N fingerprints from a document. To introduce furthermore strictness in the similarity between documents, LSH provides a technique of forming the bands of fingerprints. In this approach, some hash functions are tied together to represent them as a single hash. If two documents are highly similar, the corresponding tied-up hash functions will have the same hash in both of the documents with high probability. With this approach, the dimension is reduced further than that of Minhash sketches, as there is one hash for each band of hashes. The other remarkable advantage is the removal of false positives that might occur in Minhash due to the imperfect nature of hash functions. If r hash functions are tied up together to form b bands, where $b = N/r$, then the document with Jaccard similarity s is detected as truly similar by Minhash with probability given by $1 - (1 - s^r)^b$ [58]. The plots in Figure 5.1 shows the S-curve of the probability by which the



(a) N=200 **(b)** N=300

Figure 5.1: Plot showing the probability by which document with given Jaccard similarity will be selected as truly similar by Minhash for different number of hash functions and band sizes.

document would be detected as similar for a given Jaccard similarity between them. Figure 5.1a is for $N = 200$ fingerprints which are evenly divided into 10, 20, 40 and 100 bands. Similarly, Figure 5.1b is for 300 fingerprints evenly divided into 25, 30, 60 and 100 bands. As shown in those figures, as the band size increases the search becomes more strict. For Minhash with 200

fingerprints, which are divided into bands each containing two fingerprints, the two documents are selected as similar candidates with 50% probability if their Jaccard similarity is $\approx 10\%$. For the same scenario, banding with 20 fingerprints in each band requires the documents to have Jaccard score of at least 85% to be detected by Minhash with 50% probability. Further, the probability of being selected rises quickly after crossing the threshold Jaccard score and the documents are more likely to be selected. This way the sensitivity of Minhash can be adjusted by tuning the parameters b and r. Further, the threshold of Jaccard similarity required for Minhash to detect documents as truly similar with a 50% probability can be estimated as $(1/b)^{(1/r)}$ [58].

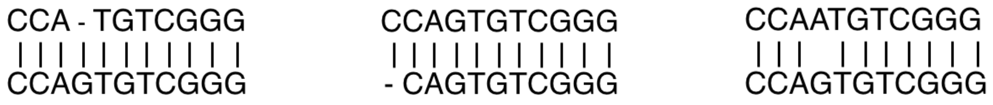
In DNNAligner, the reference genome is divided into multiple segments as discussed in Chapter 4. For each of the segments, the reads are extracted using a sliding window of fixed size and the corresponding Minhash sketches are computed for each of the reads. The sketches are indexed into the hash-table and stored in a file, which are used later while performing the alignment. While aligning the short reads from second-generation sequencing technologies, the Minhash index of the corresponding segment as predicted by the Neural Network is searched using the hash-table. The results are ranked based on the number of hash matches. Finally, local alignment using a dynamic programming algorithm is performed to compute the ground truth of alignment.

For aligning the long reads, the query string is chopped into multiple subsequences. The reason for this is due to the fact that both Neural Network and Minhash tends to perform weakly as the length of the input sequence varies than the length used for training and indexing. Normally, the long reads obtained from the third generation technologies are almost 50 times larger. After chopping, the subsequences have the same length as the one used for training and indexing. The variation of performance while varying the length of input reads are discussed in more detail in Section 7.4. After chopping, each of the subsequences is fed through the Neural Network to predict their corresponding labels. The adjacent subsequences should ideally be predicted into the same segment or the adjacent neighboring segment. However, due to the repeats in genome and inherent error in the reads, the subsequences might be predicted to different regions in the genome. The predictions from all the subsequences are aggregated to form chains, which will be later ranked before performing dynamic programming based alignment. The chains are computed by connecting the predictions for each subsequence, such that a chain starting at a subsequence having predicted segment as S_i, can only connect to another subsequence having predicted segment of either S_{i-1}, S_i or S_{i+1}. The chains are sorted based on their length, which is similar to taking a vote from all the subsequences of the long read to determine where it can be mapped. Later, for each chain, the corresponding Minhash indices are searched for their probable localities. The results of Minhash are first ranked based on their score and the longest possible chain of probable localities, as discussed earlier, are visited first for computing the local alignment. After local alignment, the alignments having a score above the threshold are reported. Finally, in the implementation of DNNAligner, the short read alignment module and long read alignment module are separated, which can be turned on/off using flags while using the tool.

Chapter 6

Local alignment

Local alignment refers to the process of matching the characters in two sequences to compute the number of edit operations required to transform one sequence into another. The edit operations are defined as insertion, deletion, and substitution of a character, which are as shown in Figure 6.1. In a sequence alignment problem, local alignment is considered as the final step before reporting any information about the alignment. Local alignment is the ground truth of comparison between two sequences. The query sequence is considered as mapped to the locus of the target sequence if the subsequence of target starting at that location has the edit operations below the permitted threshold.



(a) Insertion **(b)** Deletion **(c)** Substitution

Figure 6.1: Unit edit operations to transform one sequence to another

Let q be the query string with length m and $q_1 q_2 q_3 ... q_m$ be the characters in q. Similarly, t be the target string having length n with characters $t_1 t_2 t_3 ... t_n$. The classical dynamic programming algorithm used for alignment can be written as:

$$S_{i,j} = \min \begin{cases} S_{i-1,j-1} + \delta_{ij} \\ S_{i-1,j} + 1 \\ S_{i,j-1} + 1 \end{cases} \quad \text{where } \delta_{ij} = \begin{cases} 0; \text{ if } q_i = t_j \\ 1; \text{ otherwise} \end{cases}$$

For local alignment, the table S, having $(m + 1)$ rows and $(n + 1)$ columns, is initialized as $S_{0,j} = 0$, which comes from the fact that the query string can start anywhere in the target string and mismatches before consuming the characters in the query string should be zero. Similarly, $S_{i,0} = i$, which signifies the indel penalty of 1 for not consuming the query string. The columns in the last(m^{th}) row where the score $S_{m,j} \leq k$ are the places where the query is aligned to target with less than k edit distance. The classical dynamic programming algorithm has a time and space complexity proportional to $(m + 1) \times (n + 1)$. From the above recurrence relation, it can be observed that the value at $S_{i,j}$ is only dependent on the values at $S_{i-1,j-1}$, $S_{i,j-1}$ and $S_{i-1,j}$. The space complexity can be optimized to $O(m)$ by computing columns in the table from left to right and keeping only one column of the table in the memory. Further, Ukkonen (1985) [59] proposed an optimization which reduces the time complexity to $O(kn)$ by discontinuing the computation of $S_{i,j}$ in the dynamic programming table that leads to an edit distance of above k.

Myer's bit-vector algorithm [60] is another optimization for fast computation of the dynamic programming table. It is based on Ukkonen's approach but it formulates the dynamic programming equation in terms of deltas (differences of adjacent cells in the DP table). According to Myer's bit-vector algorithm, the dynamic programming table can be encoded as deltas using following the relation:

$$\begin{aligned}
\text{Horizontal difference, } \Delta h_{i,j} &= S_{i,j} - S_{i,j-1} \in \{-1, 0, +1\} \\
\text{Vertical difference, } \Delta v_{i,j} &= S_{i,j} - S_{i-1,j} \in \{-1, 0, +1\} \\
\text{Diagonal difference, } \Delta d_{i,j} &= S_{i,j} - S_{i-1,j-1} \in \{0, +1\}
\end{aligned} \tag{6.1}$$

The dynamic programming recurrence equations based on the deltas can be written as [60]:

$$\begin{aligned}\Delta v_{i,j} &= S_{i,j} - S_{i-1,j}\\ &= \min\begin{Bmatrix} S_{i-1,j-1}+\delta_{ij}\\ S_{i-1,j}+1\\ S_{i,j-1}+1\end{Bmatrix} - S_{i-1,j} = \min\begin{Bmatrix} S_{i-1,j-1}+\delta_{ij}\\ S_{i-1,j-1}+\Delta v_{i,j-1}+1\\ S_{i-1,j-1}+\Delta h_{i-1,j}+1\end{Bmatrix} - (S_{i-1,j-1}+\Delta h_{i-1,j})\\ &= \min\begin{Bmatrix} -1+\delta_{ij}\\ \Delta v_{i,j-1}\\ \Delta h_{i-1,j}\end{Bmatrix} + (1-\Delta h_{i-1,j})\end{aligned}$$

Similarly,

$$\Delta h_{i,j} = \min\begin{Bmatrix} -1+\delta_{ij}\\ \Delta v_{i,j-1}\\ \Delta h_{i-1,j}\end{Bmatrix} + (1-\Delta v_{i,j-1})$$

The original dynamic programming table involving $S_{i,j}$ can be obtained using the relation $S_{i,j} = \sum_{r=1}^{i} \Delta V_{r,j}$.

Further, Myer's bit-vector algorithm proposes a way to squeeze the dynamic programming computation involving integers into the bits of a machine word. As shown in the (6.1), the possible values of horizontal deltas are $\{-1, 0, +1\}$, which can be encoded using two boolean variables. Similarly, the vertical delta can be encoded using two boolean variables and the diagonal delta using one boolean variable. Hence, Myer's approach introduces the use of five boolean variables — Horizontal positive(HP), Horizontal negative(HN), Vertical positive(VP), Vertical negative(VN) and Diagonal zero(D) to encode the integer computation using boolean variables. As an example: $\Delta v = +1$ can be encoded as (VP,VN)=(true,false), $\Delta v = 0$ is encoded as (VP,VN)=(false,false) and $\Delta v = -1$ as (VP,VN)=(false,true). This way the dynamic programming computation for each cell can be squeezed into the five bits of a machine word. Further, this fact leverages parallelization in modern computers by the use of *Single Instruction, Multiple Data (SIMD)* machine instruction sets. The use of deltas in SIMD makes efficient use of SIMD registers causing the computation of dynamic programming table to be fast and parallel, disregard the length of the query and the target string. Myer's bit-vector algorithm also discussed the equivalency of bit-wise operations for replacing the recurrence relation involving integers as discussed above. Along with Myer's algorithm, there are multiple popular pieces of literature that discuss block computation of the table overcoming dependencies among the cells to enhance the use of SIMD instructions to leverage fast dynamic programming [61–63].

In DNNAligner, we are using Edlib [64], which is a C/C++ library that uses Myer's bit-vector algorithm and SIMD for fast computation of the edit distance between two sequences. The local alignment is performed for the top ranking subsequences from the reference genome as found by the Minhash. The alignment is reported for those sequences which have edit distance within the user provided threshold limit. The number of alignments to be reported is also a user-defined parameter. The alignments are reported as SAM (Sequence Alignment Map) format [65]. SAM is a popular data format for reporting sequence alignment results and it is compatible with a wide variety of downstream sequence analysis software in the pipeline.

Chapter 7

Results and Discussion

7.1 Minhash vs Deep Neural Network

One of the prime objectives of this thesis was to see if the Neural Network can provide a better hashing approach for Locality Sensitive Hashing (LSH). Minhash uses multiple hash functions and represents the document in a lower dimension using probability distribution over hash functions. Neural network internally learns the similar notion of hashing from the input sequences and, in our application, the representation learned from the sequences are used for classification of those sequences. As discussed in the Literature Review section, from the high-level point of view, the objective of both Minhash and Neural network is to narrow down the search procedure to a comparatively smaller region of the genome, so that the cost of dynamic programming is spent only in that small region. For relative comparison between them, we compared them based on sensitivity and specificity of the search. For this analysis, both the Neural network and Minhash are tested with a set of reads whose true location is known. For a given read, Neural network can predict the corresponding segments, whereas Minhash can predict the localities where the read might be aligned. For a common ground of comparison, we translate the localities predicted by Minhash into the segments (classes) as used by Neural network. Now, the problem can be defined as a multi-label classification problem, where both Minhash and Neural network can classify the reads to multiple segments (for each segment, we have a binary classification). Let Y be the true label set and Z be the predicted label set, then the precision and recall, as used in the literature of multi-label classification [66], can be defined as:

$$Precision = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Z_i|} \qquad Recall = \frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap Z_i|}{|Y_i|}$$

Because of the repeats in the reference genome and presence of the error in the aligned read, some of the localities found by Minhash might indeed be similar-enough to the input read. Therefore, to have a fair comparison, the localities found by Minhash that are very similar to the input read are not considered as false positive. For doing so, the input read is aligned with each positive segment as predicted by Minhash. The score of 0 for each matching character and 1 for each mismatch/indel is used while performing the fit alignment. The localities having a score above some threshold are not considered as false positives. The score is computed as:

$$Score = \frac{|s| - FA(s, Ref)}{|s|} \tag{7.1}$$

where, s is the input read, $|s|$ is the length of that read, Ref shows the segment Minhash has detected as positive, and FA is the fit alignment function with the aforementioned settings. As we can see, this score is a normalized score ranging from 0 to 1.

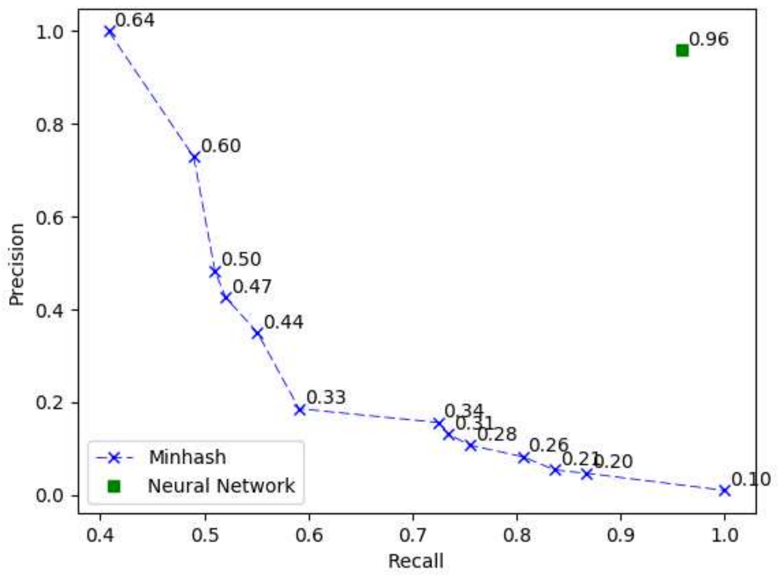


Figure 7.1: Precision and Recall for Minhash and DNN. The numbers represent Matthews correlation coefficient (MCC)

The Matthews correlation coefficient (MCC) [67] is a popularly used metric for contrasting specificity and sensitivity of a classifier. For a multi-label classification problem, MCC can be approximated as [68]: $MCC \approx \sqrt{Precision \times Recall}$

Minhash can be tuned to capture documents with Jaccard similarity above some desired threshold as discussed in Chapter 5. By tweaking the parameters of Minhash, we can obtain different Minhash predictors with variable sensitivity and specificity. Figure 7.1 shows the plot of precision vs recall for DNN and Minhash. Neural network does not have any configurable parameters so we only have one point in the plot representing DNN. For Minhash, it can be observed that recall and precision are reverse correlated and the MCC is bounded. On the other hand, the MCC score for Neural network is obtained to be 0.96, which denotes high precision and high recall. In this sense, DNN wins the task of mapping the reads to some locality over Minhash. Thus, Neural network learns a better hashing approach than Minhash.

7.2 Evaluation with Simulated Reads

For evaluation of our sequence alignment tool, DNNAligner, the reference genome of the organisms Escherichia coli and Saccharomyces cerevisiae (Yeast) were used. E.coli is a prokaryotic organism with a genome size of 4M base pairs. Yeast is a eukaryotic organism with 12M base pairs, organized into 16 chromosomes. The reference genome of those organisms was used for training a Neural network model and to build the Minhash index as discussed in previous chapters. Later, those reference genomes were used to simulate reads from different platforms using various simulators and those reads were aligned using DNNAligner for evaluation. To synthesize artificial reads, with specific error models, following simulators were used, each of which mimics a specific sequencing technology: DWGSIM [69] for simulating Illumina reads, ART [70] for simulating Roche 454 reads, SimLoRD [71] for PacBio SMRT reads and NanoSim-H [72, 73] for Oxford Nanopore reads. The performance of DNNAligner was compared against the best existing approaches in the literature: Minimap2 [49], BWA-MEM [74] and GraphMap [46].

The ground truth of the synthesized reads are known to us, so we compare our alignment result against the ground truth to compute metrics such as Recall, Precision, and Sensitivity, as used in other literature [75]. The recall is defined as the ratio of the total number of correctly aligned reads to the total number of reads. Precision is defined as the ratio of the number of correctly aligned reads to the total number of aligned reads. Sensitivity is defined as the ratio of the total number of aligned reads to the total number of reads. Further, the same set of simulated reads and the reference genome were used for alignment with Minimap2, BWA-MEM, and GraphMap to get those metrics for relative comparison of DNNAligner against them.

7.2.1 Alignment of Short Illumina reads

For aligning the reads from second generation technologies, the neural network models were trained with reads of length 200. While generating the reads, we created at least 10 copies of each read and introduced random mutations to them. For each copy, the mutation consists of insertion/deletion/substitution errors up to 10%. The hyper-parameter tuning is done by sampling random data points, as discussed in Chapter 4. Once the best hyper-parameters are discovered, the entire training set is used for training the neural network.

We simulated reads for both E.coli and Yeast genome with error-rates of `0.01`, `0.02`, `0.03`, `0.04`, and `0.05`. The simulated reads have a length of 200. For both E.coli and Yeast, the recall, precision, and sensitivity of the aligners under study were computed, which are tabulated in Table 7.1. For computing those metrics, we considered a read as correctly aligned if they are within ± 10 nucleotides from their real position.

From the Table 7.1, it can be observed that the performance of DNNAligner while aligning short reads, is quite comparable to other popular sequence alignment tools in the industry. For the results obtained using the Yeast genome, the precision of DNNAligner is higher than other tools. The sensitivity of DNNAligner is comparatively lesser but we believe it can be accounted for by training the Neural network with more noisy data points, which should account for recall score as well. Further, while creating the noisy data points, using a platform-specific error model, instead of random distribution, should address the sensitivity and recall performances. Similarly, for E.coli genome, the results are not the best but quite close to other tools.

		Escherichia coli,K-12				Saccharomyces cerevisiae (Yeast)		
Error		Recall	Precision	Sensitivity		Recall	Precision	Sensitivity
0.01	Minimap2	**0.9891**	0.9891	**1.0**		**0.9712**	0.9712	**1.0**
	BWA MEM	**0.9891**	0.9891	**1.0**		**0.9711**	0.9711	**1.0**
	GraphMap	0.9876	**0.9953**	0.9924		0.9681	0.9813	0.9865
	DNNAligner	0.9857	0.9949	0.9906		0.9609	**0.9850**	0.9754
0.02	Minimap2	**0.9890**	0.9890	**1.0**		**0.9709**	0.9709	**1.0**
	BWA MEM	**0.9890**	0.9890	**1.0**		**0.9710**	0.9710	**1.0**
	GraphMap	0.9876	**0.9955**	0.9921		0.9676	0.9813	0.9859
	DNNAligner	0.9853	0.9953	0.9899		0.9581	**0.9866**	0.9710
0.03	Minimap2	**0.9883**	0.9883	**0.999**		**0.9703**	0.9703	**0.999**
	BWA MEM	**0.9883**	0.9883	**0.999**		**0.9703**	0.9703	**0.999**
	GraphMap	0.9872	0.9953	0.991		0.9670	0.9814	0.985
	DNNAligner	0.9833	**0.9956**	0.9877		0.9535	**0.9880**	0.9650
0.04	Minimap2	**0.9888**	0.9889	**0.998**		**0.9686**	0.9687	**0.999**
	BWA MEM	**0.9888**	0.9888	**0.998**		**0.9687**	0.9688	**0.999**
	GraphMap	0.9871	0.9955	0.9915		0.9663	0.9815	0.9845
	DNNAligner	0.9847	**0.9971**	0.9876		0.9455	**0.9892**	0.9558
0.05	Minimap2	0.9879	0.9888	0.998		0.9648	0.9656	**0.991**
	BWA MEM	**0.9887**	0.9888	**0.999**		0.9657	0.9658	0.99
	GraphMap	0.9869	0.9954	0.9914		**0.9658**	0.9818	0.9836
	DNNAligner	0.9815	**0.9976**	0.9839		0.9321	**0.9904**	0.9410

Table 7.1: Recall, Precision and Sensitivity for simulated Illumina reads for E.coli and Yeast with various error-rate

7.2.2 Alignment of Roche 454, PacBio SMRT and ONT reads

For Roche 454, which is a second generation technology, we used the same model that was used for aligning Illumina reads. For this setup, we eliminated the simulated reads which are shorter than 100 nucleotides, which consists of $\approx 10\%$ of the total simulated reads. The reason for this elimination of reads is explained in the section 7.4, where we discuss about variation in performance of DNNAligner with respect to input read lengths. The reads were considered as correctly mapped if they fall within ± 10 from their true location in the genome.

For aligning the reads from third generation technologies—PacBio SMRT and Oxford Nanopore Technology (ONT), the Neural network models were trained with reads of length 500. For each reads, we created 10 mutated copies having 15% insertion/deletion/substitution errors. The reads for both E.coli and Yeast genome were simulated with the default error model provided by the corresponding simulator for PacBio and ONT as discussed previously. For PacBio and ONT reads, which are usually longer than $\approx$ 20K bases, the reads are considered as correctly mapped if they fall within ± 100 nucleotides from their original locus on the reference genome. Recall, precision, and sensitivity of different aligners are tabulated in Table 7.2.

		Escherichia coli,K-12			Saccharomyces cerevisiae (Yeast)		
		Recall	Precision	Sensitivity	Recall	Precision	Sensitivity
PB SMRT	Minimap2	0.9786	0.9794	0.9991	**0.998**	**0.999**	0.998
	BWA MEM	0.642	0.642	**1.0**	0.978	0.978	**0.999**
	GraphMap	**0.9976**	**0.999**	0.998	0.9814	0.993	0.988
	DNNAligner	0.9576	0.9578	0.9997	0.979	0.981	0.997
Nanopore	Minimap2	**0.9978**	**0.997**	0.999	0.9766	0.9811	**0.9954**
	BWA MEM	0.8485	0.8485	**1.0**	**0.9815**	**0.9820**	0.9924
	GraphMap	0.9528	0.9533	0.999	0.8156	0.8282	0.9847
	DNNAligner	0.9473	0.9485	0.9987	0.8046	0.8428	0.9536
Roche 454	Minimap2	0.988	0.988	0.99	**0.9918**	**0.9929**	0.9988
	BWA MEM	**0.9892**	0.9892	**0.999**	0.9712	0.9712	**1.0**
	GraphMap	0.9237	**0.995**	0.928	0.9681	0.9800	0.9878
	DNNAligner	0.9549	0.9917	0.9628	0.9512	0.9775	0.9730

Table 7.2: Recall, Precision and Sensitivity for simulated Roche 454, PacBio SMRT and Oxford Nanopore reads

From the Table 7.2, it can be observed that the performance of DNNAligner is quite comparable to other tools. For most of the cases, DNNAligner outperforms BWA-MEM and in general, the performance of DNNAligner is similar to GraphMap. We believe that training the neural network with noisy reads having more error rate ($> 15\%$) can make the neural network more resilient towards the error introduced by third generation technologies and will perform better.

7.3 Evaluation with real reads from Illumina

For evaluation with real reads, we used the reads sequenced with Illumina devices. For E.coli K-12 strain, we tested with the dataset published in Sequence Read Archive (SRA) with accession number (ERA000206). For Yeast, we tested with Illumina MiSeq reads from W303 strain [76]. The alignment of those reads was performed with the model that we trained for Illumina short reads, which was also used for testing the simulated short reads as discussed in the previous section. The read length of E.coli. data set is 100, while the read length for Yeast is 250. As the ground truth for those reads is not known, we compare the performance based on the sensitivity. Sensitivity is defined as the ratio of the number of aligned reads to the total number of reads. The sensitivity values for different aligners that we tested are listed in Table 7.3. As seen in the table, the performance of DNNAligner is similar to other popular tools, which shows that DNNAligner can be readily used for read world data sets also. The input reads have different read length than the neural network was trained for, yet DNNAligner can perform satisfactorily.

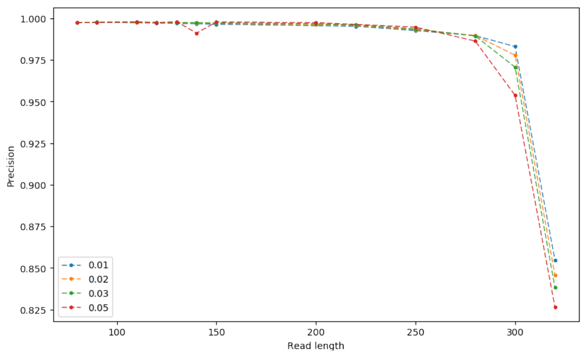
	Escherichia coli	Saccharomyces cerevisiae
BWA MEM	**0.9970**	**0.9755**
Minimap2	0.9787	0.9725
GraphMap	0.9814	0.9621
DNNAligner	0.9613	0.9656

Table 7.3: Sensitivity obtained for Illumina reads from E.coli (Accession: ERA000206) and Yeast (Strain: W303).

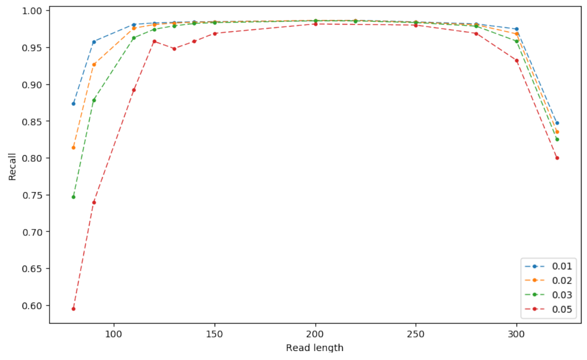
7.4 Variation in read length

From the above-discussed results, we observed that the neural network performs satisfactorily when the input reads have the same length as the reads used for training. However, the real reads from the sequencing machines are not always the same as the one that we used for training. In this section, we discuss the variation in precision, recall, and sensitivity of the neural network with varying input read length. For this experiment, we used the same model that we used for short E.coli reads, which was trained using the reads of length 200. We simulated the Illumina reads of various lengths and error rate of 0.01, 0.02, 0.03 and 0.05 and performed alignment using DNNAligner. The Precision, Recall and Sensitivity variation of the alignment with respect to read length is as shown in Figure 7.2.

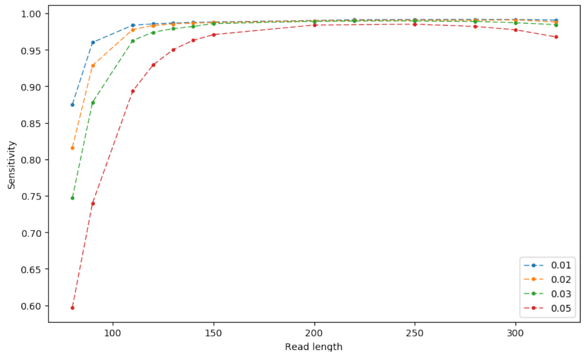
As seen in Figure 7.2c, the sensitivity is lesser when the input read length is smaller than the read length used for training the neural network. The input vector is sparse due to the less number of constituent k-mers in the smaller read and the neural network fails to predict it into segments.



(a) Variation in Precision



(b) Variation in Recall



(c) Variation in Sensitivity

Figure 7.2: The plot showing variation of Precision, Recall and Sensitivity of DNNAligner with respect to varying read lengths.

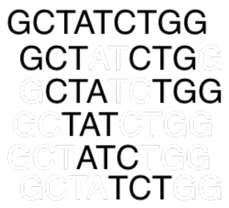
As the read length increases, the sensitivity gradually increases too, which is due to the fact that the input vector begins to contain enough features to make the prediction as there are more constituent k-mers. On the other hand, as the read length increases, the precision gradually decreases since the network tends to make incorrect predictions, which can be seen in Figure 7.2a. The incorrect predictions mislead the alignment into false segments, which results in the reported alignment to be false. Hence, the DNNAligner works best when the input reads have a similar length as the reads used for training the neural network. But still, there is a wide band of read lengths where DNNAligner performs satisfactory, which can be seen in the plot of variation in Recall as shown in Figure 7.2b.

7.5 Capturing longer strings using k-mer

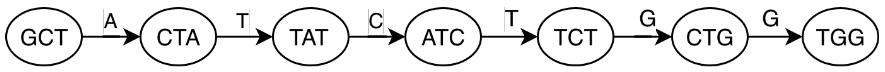
As observed in the above results and in the comparison of LSTM and MLP networks, using a relatively simple model of a deep neural network can predict the reads with very high accuracy. In this section, we discuss some intuition about how an MLP can result in such a model, even with small k ($k = 7$ in our case).

In the vector representation of the reads, the reads are broken down into constituent k-mers. The vector is encoded with either 1 or 0 to denote either presence or absence of a given k-mer out of all 4^k possible k-mers. While breaking them, the information about the relative position of the k-mers with respect to each other seems to be lost. But one obvious fact is that, given a list of constituent k-mers, one can compute the original sequence by building a de Bruijn graph. De-Bruijn graph is a directed graph where the vertices are the k-mers and an edge is drawn between two vertices if there is a $k - 1$ overlap between them. This idea is illustrated in Figure 7.3. In the Figure 7.3a, the original sequence `GCTATCTGG` is broken into its constituent 3-mers – `GCT`, `CTA`, `TAT`, `ATC`, `TCT`, `CTG` and `TGG`. Using those 3-mers, a de Bruijn graph is built as shown in Figure 7.3b. The node `TAT` and `ATC` has a $k - 1$ overlap of `AT`, hence they are connected with an edge. Similarly, edges are drawn between nodes for all the overlaps. Finally, a traversal in de Bruijn graph can reconstruct the original sequence. In the presence of repeated k-mers, the de Bruijn graph will contain ambiguous paths of traversal, but still the original sequence can be partially reconstructed. The neural network can learn this notion and can make correspondence between some groups of k-mers occurring in the input sequence to the segment that they originally belong to. Hence, the k-mer composition is rich enough to represent the longer originating sequence.

Another observation from our experiments is that $k = 7$ worked best for us. If k is large enough, the probability of random occurrence of such groups of k-mers, as discussed above, becomes lower, which makes a group of k-mers specific to some segments in the genome. Hence, increasing the size of k-mer increases the specificity of the model. On the other hand, the size of vector representation increases exponentially by increasing k, which adds a lot of memory and computation overhead. Besides that, larger vector representation causes vectors to be sparse, too



(a) 3-mers of the sequence `GCTATCTGG`



(b) de Bruijn graph constructed from the constituent 3-mers

Figure 7.3: Re-construction of the original sequence from the k-mers using de Bruijn graph.

specific and might cause overfitting in the training data, which decreases the ability to make a good prediction in general unseen data. Some values of k ($k < 7$ and $k > 7$) were evaluated but we observed that $k = 7$ works best for E.coli and Yeast genome.

Chapter 8

Conclusion and Future work

Although the sequence alignment problem has been researched for a few decades and lots of breakthroughs has been made, there has not been much research about exploring the options of a Deep Neural Network for sequence alignment. Our aim with this project was to explore the pattern recognition capability of the Deep Neural Network in the sequence alignment. We used the reference genome of E.coli (Escherichia coli) and Yeast (Saccharomyces cerevisiae) for our studies. We trained a neural network and evaluated our approach with real and simulated data sets. We also compared the performance of our approach to other popular approaches currently used in the bioinformatics community.

We showed that the Deep Neural Network can provide some notion of hashing similar to Minhash, which is, in fact, superior in performance. The classification of the reads to corresponding segments is performed with high recall and precision by the Neural network, while Minhash is limited by the trade-off between sensitivity and specificity. This idea is not limited to the sequence alignment problem and can be utilized in other genomic sequence analysis problems which require determining similar sequences. In the future, we would like to explore other problems in bioinformatics which suits these use cases.

In our study, we observed that the sequence alignment using the neural network, by narrowing down the search in a particular region of the genome, is quite comparable to other heuristics based approaches. Our tool performed better than other popular tools in terms of precision in simulated data sets of Yeast (Saccharomyces cerevisiae). We also observed that DNNAligner is resilient to some extent of variation in read length than the one it was trained for, but we would like to explore other neural network architectures that are tolerant to the variation in input read length. One of them being LSTM, we would like to explore into it further. Also, we have evaluated our tools for smaller size genomes, but in future, we would like to use our approach for mammalian size genome like human genome.

Bibliography

[1] Frederick Sanger, Steven Nicklen, and Alan R Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[2] Wikipedia contributors. DNA — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=DNA&oldid=861568134, 2018. [Online; accessed 6-October-2018].

[3] Frederick R Blattner, Guy Plunkett, Craig A Bloch, Nicole T Perna, Valerie Burland, Monica Riley, Julio Collado-Vides, Jeremy D Glasner, Christopher K Rode, George F Mayhew, et al. The complete genome sequence of Escherichia coli K-12. *science*, 277(5331):1453–1462, 1997.

[4] International Human Genome Sequencing Consortium et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860, 2001.

[5] Adrian V Dalca and Michael Brudno. Genome variation discovery with high-throughput sequencing data. *Briefings in bioinformatics*, 11(1):3–14, 2010.

[6] Shirley Pepke, Barbara Wold, and Ali Mortazavi. Computation for ChIP-seq and RNA-seq studies. *Nature methods*, 6(11s):S22, 2009.

[7] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 33(6):623, 2015.

[8] Shawn J Cokus, Suhua Feng, Xiaoyu Zhang, Zugen Chen, Barry Merriman, Christian D Haudenschild, Sriharsa Pradhan, Stanley F Nelson, Matteo Pellegrini, and Steven E Jacobsen. Shotgun bisulphite sequencing of the Arabidopsis genome reveals DNA methylation patterning. *Nature*, 452(7184):215, 2008.

[9] Elaine R Mardis. Anticipating the $1,000 genome. *Genome biology*, 7(7):112, 2006.

[10] Jay Shendure and Hanlee Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135, 2008.

[11] Mehdi Kchouk, Jean-François Gibrat, and Mourad Elloumi. Generations of sequencing technologies: From first to next generation. *Biology and Medicine*, 9(3), 2017.

[12] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC genomics*, 13(1):341, 2012.

[13] Kim R Rasmussen, Jens Stoye, and Eugene W Myers. Efficient q-gram filters for finding all ε-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006.

[14] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[15] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[16] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[17] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[18] W James Kent. BLAT-the BLAST-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[19] William R Pearson. [5] rapid and sensitive sequence comparison with FASTP and FASTA. 1990.

[20] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[21] Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II: Highly sensitive and fast homology search. *Journal of bioinformatics and computational biology*, 2(03):417–439, 2004.

[22] Gregory Kucherov, Laurent Noé, and Mikhail Roytberg. Multiseed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(1):51–61, 2005.

[23] Hao Lin, Zefeng Zhang, Michael Q Zhang, Bin Ma, and Ming Li. ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

[24] Colin Meek, Jignesh M Patel, and Shruti Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 910–921. VLDB Endowment, 2003.

[25] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

[26] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

[27] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *International Workshop on Algorithms in Bioinformatics*, pages 449–463. Springer, 2002.

[28] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

[29] Steve Hoffmann, Christian Otto, Stefan Kurtz, Cynthia M Sharma, Philipp Khaitovich, Jörg Vogel, Peter F Stadler, and Jörg Hackermüller. Fast mapping of short sequences with

mismatches, insertions and deletions using index structures. *PLoS computational biology*, 5(9):e1000502, 2009.

[30] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[31] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[32] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[33] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[34] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009.

[35] Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

[36] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[37] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[38] Nawar Malhis, Yaron SN Butterfield, Martin Ester, and Steven JM Jones. SliderâĂŤmaximum use of probability information for alignment of short sequence reads and SNP detection. *Bioinformatics*, 25(1):6–13, 2008.

[39] Nawar Malhis and Steven JM Jones. High quality SNP calling using Illumina data at shallow coverage. *Bioinformatics*, 26(8):1029–1035, 2010.

[40] Stephen M Rumble, Phil Lacroute, Adrian V Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

[41] Kim R Rasmussen, Jens Stoye, and Eugene W Myers. Efficient q-gram filters for finding all ε-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006.

[42] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC bioinformatics*, 13(1):238, 2012.

[43] Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. A local chaining algorithm and its applications in comparative genomics. In *International Workshop on Algorithms in Bioinformatics*, pages 1–16. Springer, 2003.

[44] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming I: linear cost functions. *Journal of the ACM (JACM)*, 39(3):519–545, 1992.

[45] Gene Myers. Efficient local alignment discovery amongst noisy long reads. In *International Workshop on Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

[46] Ivan Sović, Mile Šikić, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nature communications*, 7:11307, 2016.

[47] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and complexity of sequences 1997. proceedings*, pages 21–29. IEEE, 1997.

[48] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

[49] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 1:7, 2018.

[50] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[51] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[52] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[53] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[54] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[55] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol

Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[56] Ondrej Chum, James Philbin, Andrew Zisserman, et al. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, volume 810, pages 812–815, 2008.

[57] Shumeet Baluja and Michele Covell. Audio fingerprinting: Combining computer vision & data stream processing. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 2, pages II–213. IEEE, 2007.

[58] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[59] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[60] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[61] Andrzej Wozniak. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, 13(2):145–150, 1997.

[62] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[63] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2006.

[64] Martin Šošić and Mile Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[65] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[66] Mohammad S Sorower. A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis*, 18, 2010.

[67] Brian W Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[68] Sinan Uğur Umu and Paul P Gardner. A comprehensive benchmark of RNA–RNA interaction prediction tools for all domains of life. *Bioinformatics*, 33(7):988–996, 2017.

[69] Nils Homer. Dwgsim (2017). *URL https://github.com/nh13/DWGSIM*.

[70] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2011.

[71] Bianca K Stöcker, Johannes Köster, and Sven Rahmann. Simlord: simulation of long read data. *Bioinformatics*, 32(17):2704–2706, 2016.

[72] Chen Yang, Justin Chu, René L Warren, and Inanç Birol. Nanosim: nanopore sequence read simulator based on statistical characterization. *GigaScience*, 6(4):1–6, 2017.

[73] Chen Yang Karel Brinda. Nanosim-h (version 1.1.0.4). *URL http://doi.org/10.5281/zenodo.1341249*.

[74] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[75] Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324, 2012.

[76] Markus Ralser, Heiner Kuhl, Meryem Ralser, Martin Werber, Hans Lehrach, Michael Breitenbach, and Bernd Timmermann. The Saccharomyces cerevisiae W303-K6001 cross-platform genome sequence: insights into ancestry and physiology of a laboratory mutt. *Open biology*, 2(8):120093, 2012.