THE UNIVERSITY OF HONG KONG

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Final Report *of* ELEC4848: Senior Design Project

Row-invariant Convolutional Neural Network with Applications to Bioinformatics

by *Yafei Xue* (UID: 3035142405)

under the supervision of *Prof. Lam Tak-Wah*

TABLE OF CONTENTS

ABSTRACT

Variant-calling is a crucial step in many forms of bioinformatics data analysis. Recently, Google's *DeepVariant* has applied image classification networks to variant-calling and achieved promising results. Nevertheless, the large size of the network has limited its application to the fields where sufficient amount of data is available. However, by exploiting the row-invariant property of the sequence alignment data, a much more compact network with a fewer number of trainable parameters can be built, which requires a much smaller amount of training data and a much shorter training time.

The study aims to design and implement a row-invariant convolutional neural network for variant-calling problems. A prototypical variant-calling problem has been designed, on which row-invariant convolutional neural network's superior performance and accuracy has been verified. A variant-calling network has been designed, implemented and optimized, nevertheless, due to insufficient time, fine-tuning and finalization of the network are excluded from the scope of this study. Still, the current results could bring insights for future work, and with appropriate optimization and tuning, the performance and accuracy of the row-variant convolutional neural network could be expected to outperform the existing tools on variant-calling problems.

Index terms: Variant-Calling, Row-Invariant Convolutional Neural Network, Large-Scale Bioinformatics Data

Acknowledgement

I would like to offer my sincere gratitude to all the people who have helped me with this study.

I would like to thank my supervisor, Prof. Tak-Wah, who has provided valuable instructions and resources for this study. I would also like to thank Dr. Ruibang Luo and Simon Wong, who have guided me through the basics of research.

I also need to say a thank-you to other lab peers who have kindly offered their helps. Wai Chun Law, who has offered great suggestions for the optimization of my data processing algorithm. Ou Min, Ricky and Ken, who have helped me with *tensorflow* installation issues. And Alex and Chase, who have been fun lunch partners when we were all busy in the lab.

List of Figures and Tables

Abbreviations

BASEQ	Base Quality Score
CNN	Convolutional Neural Network
GPU	Graphics Processing Units
MAE	Mean Average Error
MAPQ	Mapping Quality Score
NIST	National Institute of Standards and Technology
RMSE	Root Mean Square Error
SAM	Sequence Alignment Map

1 Introduction

Variant-calling, the process of determining nucleotide differences between a genome or transcriptome and some reference at given positions, is an important step in many forms of bioinformatics data analysis. DNA sequencing is one of the major applications of variant-calling. In DNA sequencing, the original DNA sequence is chopped off into many short sequences know as reads, whose exact sequencings are determined via biochemical approaches. These reads are latter mapped back to a referential DNA sequence, and the alignment data is scanned for candidate variant sites from the reference; finally, the nucleotide differences between the sequenced DNA and the referential DNA are determined by variant-calling. Both the sequencing and the mapping processes are subjected to various errors, which introduces uncertainties to the variant-calling process.

Recently, Google has shown that a deep convolutional neural network, which learns the statistical relationship between the sequence alignment data and ground-truth genotypes, has achieved promising results in variant-calling and outperformed the traditional statistical approaches[1]. The approach, called *DeepVariant*, encodes the sequence alignment data into a picture and trains an image classification neural network to determine the corresponding genotype of the variant site.

However, with over 20M parameters, the application of *DeepVariant* is currently limited to the fields where sufficient data is available[1]. By exploiting the row-invariant property of sequence alignment data, this project seeks to build a row-invariant convolutional neural network with a much fewer number of trainable parameters than the *DeepVariant* network, which thus requires a smaller amount of training data and a shorter training time.

The scope of this project includes a study of prototypical variant-calling problems, bioinformatics data processing, as well as the design, implementation, and partial optimization of a row-invariant convolutional neural network for the variant-calling process. Due to time limitation, fine-tuning and finalization of the network are excluded from the scope of this project. Nevertheless, I hope the current work could provide insights into the design and application of row-invariant convolutional neural networks on the variant-calling process, as well as offering possible directions for future works.

The report is structured as follows. Section 2 describes the theoretical basis of designing a row-invariant convolutional neural network. Section 3 introduces the design and implementation of prototypes, as well as comparing the performances of networks with different architectures on the prototypical problems. Section 4 explains the data processing process, including remarks on Python code optimization on large-scale bioinformatics data. Section 5 shows the design, implementation, and optimization of the variant-calling network, as well as offering a glimpse of the future work. Finally, Section 6 presents the conclusion.

2 Theoretical Basis of Designing a Row-invariant Convolutional Neural Network

The structure of a row-invariant convolutional neural network derives from that of a convolutional neural network (CNN). CNN's are a class of neural networks with successful applications in 2D image processing. A CNN usually consists of three types of layers, namely 1) convolutional layers, which perform 2D convolution, 2) subsampling layers, which conduct 2D maximum or average pooling, and 3) fully-connected layers. A convolutional layer takes advantage of the 2D structure of the input images, in which the hidden units are only connected to a small area of the input and convolve with the whole image to produce the output; therefore, a CNN requires far less trainable parameters than a fully-connected neural network, and thus requires a smaller amount of training data, as well as a shorter training time. Hence, CNN's can achieve superior performances comparing to fully-connected networks in various image-related applications.

Although DeepVariant has achieved promising results in applying CNN's to variant-calling problems, its performance may be further improved by fully utilizing the properties of sequence alignment data. Unlike images, sequence alignment data has the property of being row-invariant: that is, the rows in a sequence alignment data are interchangeable. The design of a row-invariant CNN aims to exploit this row-invariant property by introducing 1D convolution and 1D pooling to further reduce the number of trainable parameters in the network. Three feature operations were designed to realize a row-invariant CNN, namely row-invariant convolution, row-invariant subsampling and row-to-row convolution.

The process of conducting row-invariant convolution is illustrated in Fig 2-1. The input matrix is of the size $m \times n$, where m denotes the number of columns and n that of rows. The row-invariant convolutional layer takes a 1D kernel with the length $a \ll m$, which convolves with each row respectively to produce a feature map with the size $(m - a + 1) \times n$. Replacing a 2D kernel with a 1D kernel, a row-invariant convolutional layer requires a much smaller number of parameters compared to a convolutional layer by reducing the filter size from $a \times a$ to a.

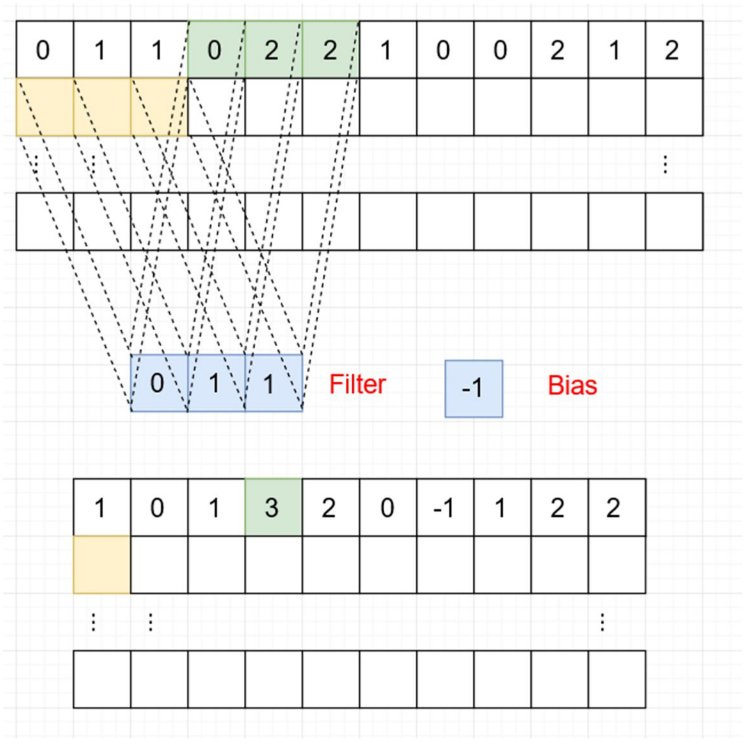

Fig 2-1 Illustration of Row-Invariant Convolution

(with $m = 12, a = 3$)

Fig 2-2 demonstrates the process of row-invariant subsampling. A row-invariant subsampling layer performs average or maximum pooling over p contiguous regions in each column of the input image. This operation is analogous to the subsampling operation in CNN's.

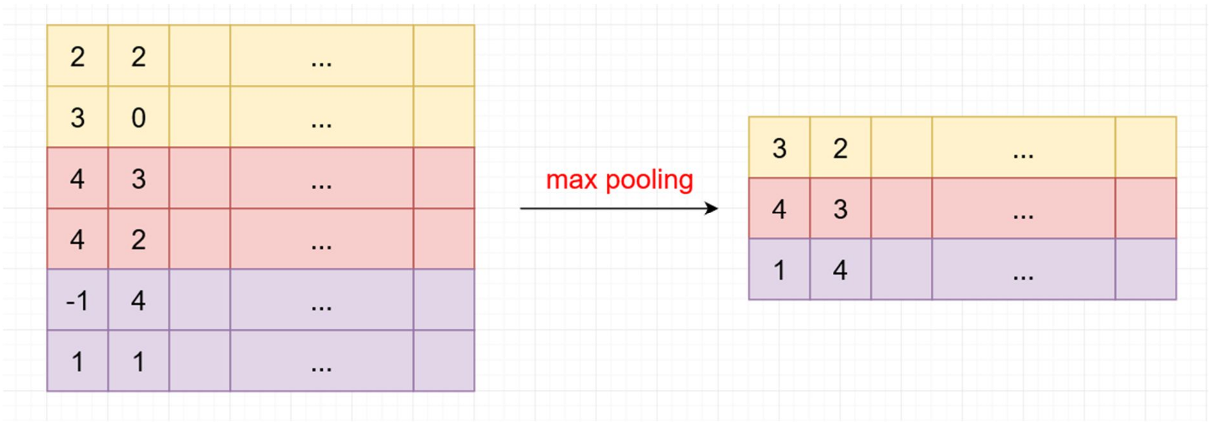

Fig 2-2 Illustration of Row-Invariant Subsampling

Fig 2-3 serves as an illustration of row-to-row convolution, which extracts the features relevant to the relationship between rows. A row-to-row convolutional layer performs row-invariant convolution on the concatenated rows in a 1-vs-all manner. In Fig 2-3, the input matrix is with the size $m \times n$; the matrix is duplicated for n times, producing a 3D matrix with the dimensions $m \times n \times n$. The 3D matrix is further duplicated, rotated, and concatenated with the original 3D matrix. The resulting 3D matrix contains the permutations of every two rows. The concatenated matrix is reshaped into a 2D matrix with the dimensions $m \times n^2$, on which a row-invariant convolution is performed to extract the features. Row-to-row convolution is performed to study the relationship between the rows in sequence alignment data as a step of the variant-calling process.

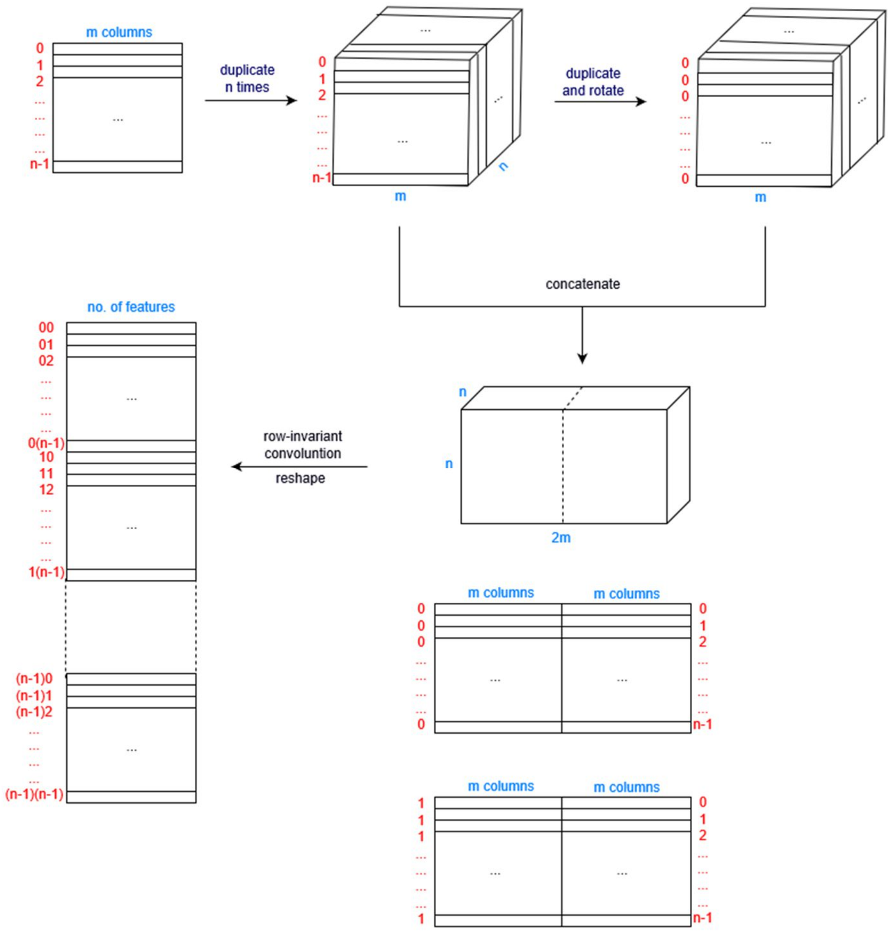

Fig 2-3 Illustration of Row-to-Row Convolution

The three feature operations have played vital roles in the design of a row-invariant CNN.

3 Design, Implementation, and Performance of Prototypes

In this study, a set of prototypical problems was designed to validate the three feature operations of row-invariant CNN's and test the performance of row-invariant CNN's on variant-calling problems. For simplicity, only two of the prototypical problems are introduced to explain the application of row-invariant CNN's on variant-calling problems and compare the performance of row-invariant CNN's against that of CNN's and fully-connected neural networks.

Section 3.1 presents the design, implementation, and performance of a row-invariant CNN on a feature prototypical variant-calling problem. The performances of a row-invariant CNN and a CNN for the problem are compared, but due to the hardware limitation, it's impossible to implement a fully-connected network and test its performance on the problem. Therefore, in Section 3.2, a simplified prototypical problem is introduced, and the performances of a row-invariant CNN, a CNN and a fully-connected neural network on the problem are compared and analysed.

3.1 The Prototypical Variant-Calling Problem

The prototype is a simplified version of real-world variant-calling problems and is designed to verify the applicability and test the performance of row-invariant CNN's on variant-calling. As will be discussed in greater details in Section 4.1, in real-world applications, the sequence alignment data contains four pieces of information relevant to the variant-calling process, namely 1) nucleotide sequences, 2) base quality scores, 3) mapping quality scores and 4) strands, the prototype only considers the nucleotide sequences for simplicity. The problem size is also reduced for computational convenience. In the prototype, the four types of nucleotide A, C, G, and T are represented by one-hot, four-bit vectors, and a null nucleotide is represented by a zero-hot, four-bit vector. A null nucleotide may indicate a deletion or a lack of information. Each row represents a read. The likelihood of each nucleotide at the variant-calling site, from which the genotype can be derived, is estimated from the probability that the rows are correctly mapped, which is further assumed to be determined by to the extent to which the row is the same as other rows. If a read is the same as most of the other reads in the sequence alignment data, then we could reasonably assume that the read is correctly mapped, and vice versa.

Following the above simplifications and assumptions, the prototypical problem states as follows, with its illustration presented by Fig 3.1-1:

Consider a matrix with 30 rows. Each row has a tag and 20 data, and each tag or datum is represented by a 4-bit, one-hot or zero-hot vector. Consider the number of rows with the same data parts (all 20 pairs of vectors being equal) and assign each row with a score. For example, for the first row, if there are 8 other rows with the same data parts as the first row, the row is assigned with a score of 9. Likewise, calculate the scores of all rows. For each tag (excluding the zero-hot tag), calculate the quotient by the sum of all scores corresponding to the tag and the sum of all scores. The four quotients are the output.

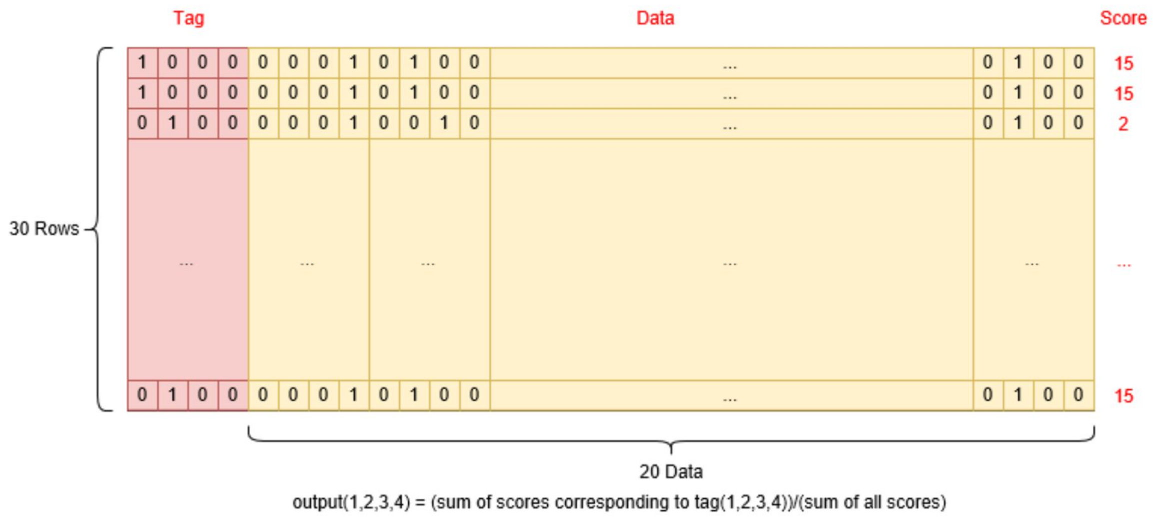
Tag
Data
Score
30 Rows
20 Data
15
15
2
15
output(1,2,3,4) = (sum of scores corresponding to tag(1,2,3,4))/(sum of all scores)

Fig 3.1-1 Illustration of the Prototypical Problem

In the prototypical problem, each row corresponds to a read in the sequence alignment data, and each 4-bit vector a nucleotide. The “tag” vector corresponds to the nucleotide at the candidate variant site and the “data” vectors to the nucleotides at the surrounding sites. In a variant-calling problem, the genotype of the candidate variant site can be determined from the likelihoods for the four types of nucleotides A, C, G, and T at the site, which is estimated from the probabilities that the reads are correctly mapped. If a read is more likely to be correctly mapped, its assertion of the nucleotide at the candidate variant site also carries more credibility. The “score” of each row, which represents the number of rows with the same data parts as this row (including the row itself), approximates such probability. Note that a null nucleotide at the candidate variant site indicates a deletion or a lack of information and need not be considered in a genotype call.

A row-invariant CNN is constructed to solve the problem, its architecture shown in Fig 3.1-2. Since scores are derived solely from the data, the data and tag parts are first split. A row-to-row convolution is performed on the data part to retrieve information about the relationship between the data parts of rows (whether they are equal), which determines the scores of the rows. Recall from Section 2 that a row-to-row convolution squares the number of rows in the input matrix; thus, a row-invariant average pooling is required to reduce the number of rows back to 30 while preserving the feature information for the convenience of later operations. Finally, since we’re interested in the relationship between the scores and corresponding tags, the processed data part and the tag part are concatenated and passed through several row-invariant convolutional layers and fully-connected layers to obtain the result.

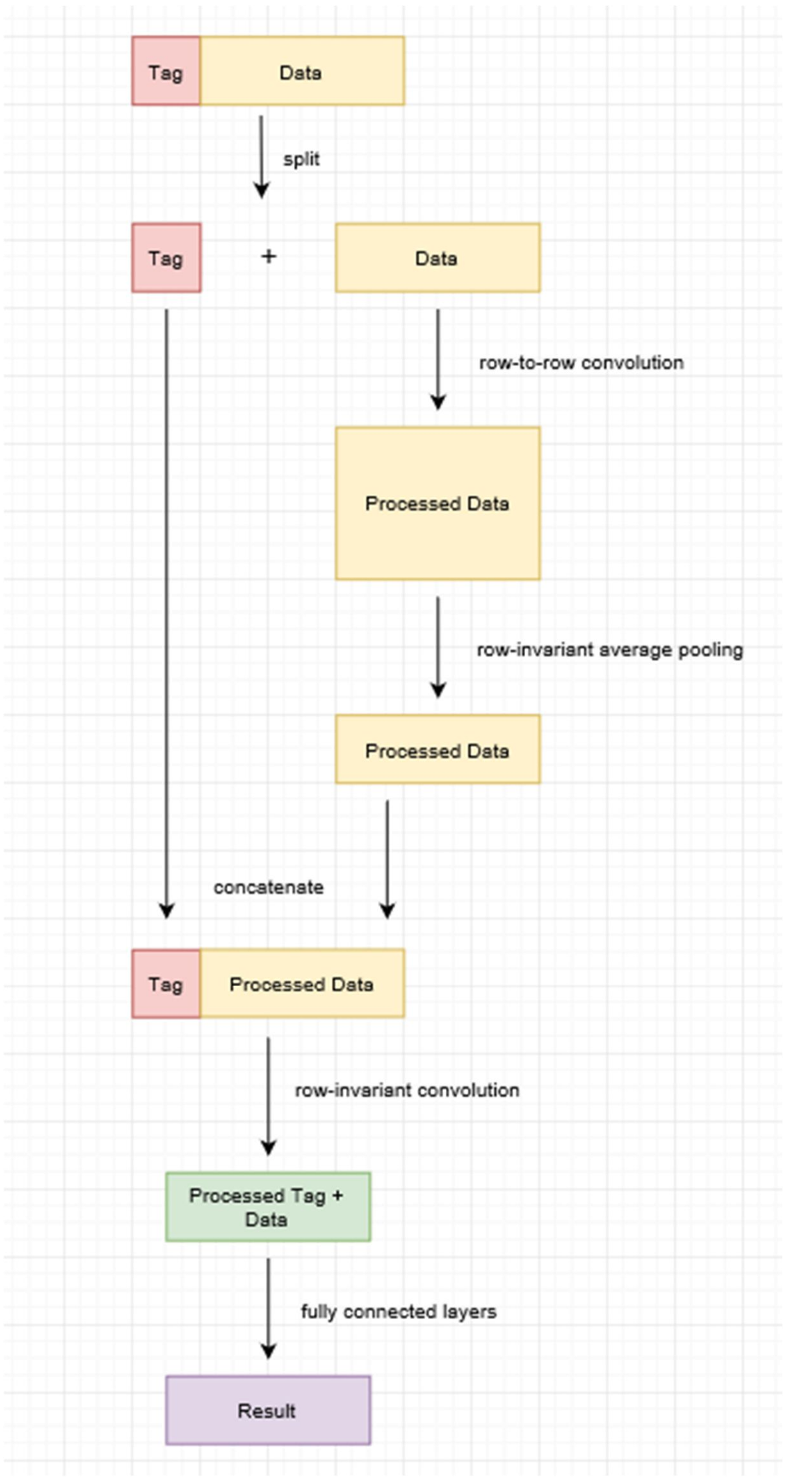

Fig 3.1-2 Architecture of the Prototypical Network

The prototype programme was written in *Python 3.6*, and the neural network was implemented with *keras* on *tensorflow* backend. *Keras* is a high-level neural network API, whose language is concise and thus enables fast implementation. Able to deploy the computation to multiple GPU units, the *tensorflow* framework exploits the programming parallelism, which could accelerate the network training tremendously.

The network was trained on 2M randomly generated data, using Adam optimizer with a default learning rate of 0.001. A dropout layer was attached to improve the robustness of the model. The network has 30,662 trainable parameters and could achieve a mean absolute error (MAE) of 0.0005 and root mean square error (RMSE) of 0.0026 within five hours of training. The filter sizes used in both the row-to-row convolution and the row-invariant convolution are $a = 3$.

Table 3.1-3 compares the performance of a row-invariant CNN and a CNN on the prototype problem. A filter size of 3×3 is applied for the CNN. Both the networks were trained on 2M randomly generated data for 5 hours. The CNN has 202,752 parameters and could achieve an MAE of 0.0396 after the training. Hence, the row-invariant CNN requires a much fewer number of parameters and could achieve a significantly lower error than the CNN on the prototypical variant-calling problem, which has verified the superior performance of the row-invariant CNN on the prototypical variant-calling problem.

	Amount of Data	Training Time	No. of Parameters	MAE
CNN	2M	5 hours	202,752	0.0396
Row-invariant CNN			30,662	0.0005

Table 3.1-3 Comparison between a CNN and a Row-Invariant CNN on the Prototypical Problem

Due to the hardware limitation, I did not choose to implement a fully-connected neural network to solve the above problem. Instead, I chose to design a simplified problem as in Section 3.2, to which a fully-connected neural network, a CNN, and a row-invariant CNN are applied and their performance compared.

3.2 A Simplified Prototypical Problem for Performance Comparison between Row-Invariant CNN's, CNN's and Fully-Connected Neural Networks

Due to the hardware limitation, a simplified prototypical problem was designed to compare the performances of row-invariant CNN's, CNN's and fully-connected neural networks. Only the first of the three feature operations of row-invariant CNN's, row-invariant convolution, will be used in the implementation of the row-invariant CNN on the simplified prototypical problem. The problem states as below, whose illustration presented by Fig 3.2-1:

Consider a matrix with 25 rows. Each row has a tag and two data, and each tag or datum is represented by a 4-bit, one-hot or zero-hot vector. Consider all the equal one-hot vectors pairs in the data part of each row (if both data are zero-hot vectors, it is not considered as an equal one-hot vector pair). For each of the four tags (excluding the zero-hot tag), calculate the percentage of equal one-hot vector pairs corresponding to the tag over all the equal one-hot vector pairs. The output is the four percentages.

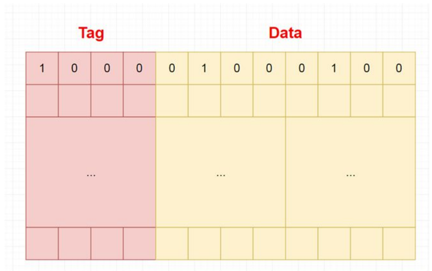

Fig 3.2-1 Illustration of the Prototypical Problem

In this problem, we are only interested in the relationship between elements within the same row but not the relationship between rows, so among the three feature operations of row-invariant CNN's, only the row-invariant convolution needs to be conducted. Three neural networks were implemented to solve the problem: a fully-connected neural network, a CNN with a filter size of 3×3, and a row-invariant CNN with a filter size of 3. Their performances are presented in Fig 3.2-2. As shown by the figure, with the same amount of training data and training time, the row-invariant convolutional neural network requires far less number of parameters and could achieve much better performance than the fully connected neural network.

	No. of Parameters	MAE	Amount of Training Data	Training Time
Fully-Connected Neural Network	~126,000	0.15	100K	5h
CNN	1,310	0.04		
Row-invariant CNN	243	<0.01		

Table 3.2-2 Comparison between a fully-connected network, a CNN and a Row-Invariant CNN on the Prototypical Problem

To summarize, the prototypes have validated the three feature operations of row-invariant CNN's and justified the effectiveness and efficiency of row-invariant CNN's in solving variant-calling problems. In Section 5, the design of the neural network for real-world variant-calling problems will be based on the architecture of the network in Section 3.1.

4 Large-Scale Bioinformatics Data Processing

In this project, the sequence alignment data used is retrieved from a human genome (chromosomes 1-22) with the National Institute of Standards and Technology (NIST) ID HG002. The raw data is stored in the .bam format, and pre-processing is required to turn the data into a suitable form for the network training. The dataset has ~213M samples, with each sample containing an average of over 30 reads, each with a length of 148 bases.

The enormous scale of the dataset has introduced great difficulty to the data processing. Three techniques, namely algorithm optimization, data compression and parallel computation, have been adopted to accelerate the code for data processing. With these techniques, the time required for data processing has been reduced from over 280 days to ~10 days.

This section is structured as follows. Section 4.1 presents an overview of the dataset. In Section 4.2, the code optimization techniques to accelerate the data processing are discussed.

4.1 Overview of the Sequence Alignment Data

The sequence alignment data used in this study is generated from a human genome (chromosomes 1-22, excluding the X and Y chromosomes) with the National Institute of Standards and Technology (NIST) ID HG002, where the raw data is stored in BAM format. The dataset contains ~213M candidate variant sites, and each site is covered by an average of over 30 reads. The read length is 148 bases.

The BAM file contains five pieces of information relevant to variant-calling:

1) The read sequence, represented by a string of the four types of nucleotides A, C, G and T.

2) The base quality score sequence, represented by an array of quality scores corresponding to the bases in the read sequence. The base quality score is an integer from 0-30, indicating the likelihood that the base read is correct. The meaning of base quality scores will be discussed in greater details in Section 5.

3) The mapping quality score, represented by an integer from 0-60, indicating the likelihood that the mapping of the read is correct. The meaning of mapping quality scores will also be discussed in greater details in Section 5.

4) The strand, represented by an integer being 0 or 1. The strand gives the direction of the read (3' or 5' end).

5) The cigar string, which gives information about how a read aligns with the reference. A cigar string consists of one or several parts, where each part contains a number and a letter. The letter corresponds to a label and the number corresponds to the number of bases the label applies to. The most common types of labels are:

> M – Match. It can either be a correct match or a mismatch. It means the nucleotides are present in both the read and the reference.
>
> I – Insertion. The corresponding piece is inserted. The nucleotides are present in the read but not the reference. The inserted piece is disposed of in data processing since the inserted nucleotide cannot be efficiently represented and carries little information relevant to variant-calling.

D – Deletion. The corresponding piece is deleted. The nucleotides are present in the reference but not the read. The deleted nucleotides are represented as null nucleotides with zero base quality scores in the matrix.

S – Soft Clipping. Soft clipping usually occurs at the beginning or the end of a read. The measured piece labelled with “S” is not accurate and discarded in data processing.

The cigar string gives crucial information about how to process the data, though does not directly appear in the processed dataset.

Like in Section 3, in the processed data, each base is represented by a 4-bit, one-hot vector corresponding to the type of the nucleotide A, C, G, or T, and a null nucleotide is represented by a 4-bit, zero-hot vector. A null nucleotide may indicate a deletion, or a base not covered by the read. The correspondence between the nucleotide and the 4-bit vector is shown by Fig 4.1-1.

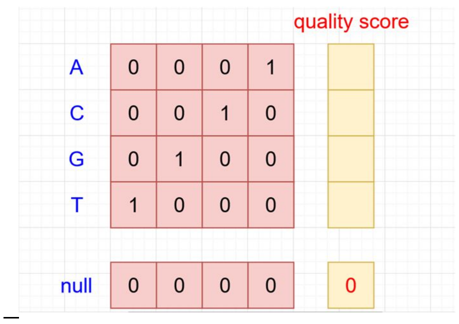

Fig 4.1-1 Correspondence between the Nucleotides and 4-bit Vectors

The sequence alignment data is organized into the matrix form as in Fig 4.1-2. Each row corresponds to a read (may be a null read if the number of reads is less than the maximum number of reads), which consists of (2*max_read_length - 1) units, where max_read_length gives the length of a read, which being 148 in our case. Each unit is comprised of a 4-bit vector corresponding to the nucleotide type or a null nucleotide, and an integer corresponding to the base quality score, or a zero when the corresponding score is not available. In each row, there are two additional pieces of information; one being the mapping quality score of the read, represented by an integer, and the other one the strand of the read, represented by an integer being 0 or 1. The number of rows is the maximum number of reads. For convenience, a maximum of 100 reads for each candidate variant site is adopted for the data processing; that is, if a site is covered by more than 100 reads, only the 100 reads with maximum mapping qualities (whose meaning will be explained later) will be considered in the variant-calling process.

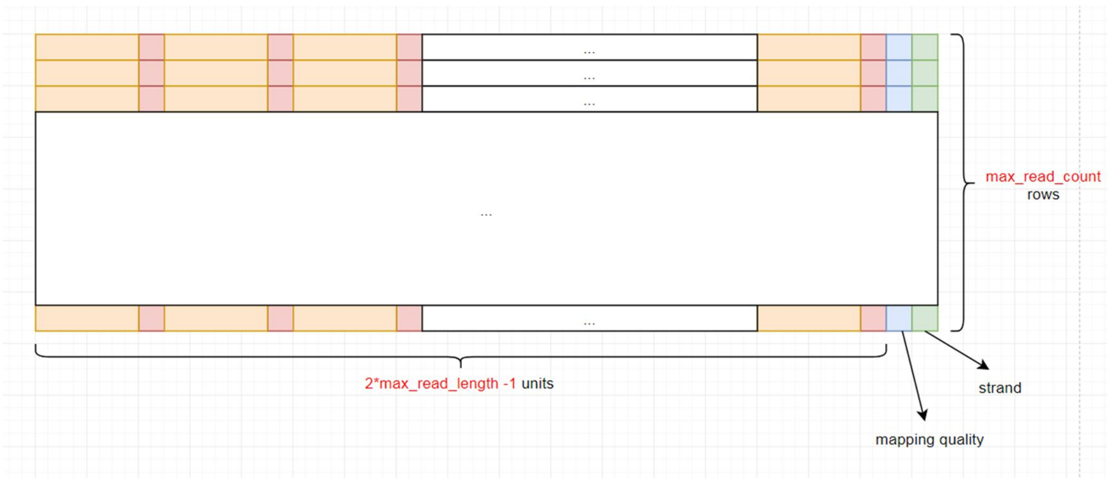

Fig 4.1-2 Illustration of the Sequence Alignment Data in Matrix Form

The scripts for data processing was written in *Python 3.6*, with the BAM file processing done by *pysam*.

4.2 Python Code Optimization for Large-Scale Bioinformatics Data

Bioinformatics datasets often come in tremendous sizes. The dataset used in this project contains ~213M samples, and each sample takes 590KB before the compression, which gives an enormous size of ~125.8TB in total. The large scale of the dataset has introduced great difficulty to the data processing. To accelerate the process, three techniques have been adopted, namely algorithm optimization, data compression, and parallel computation. With these techniques, the time consumption for data processing can be reduced from over 280 days to ~10 days.

4.2.1 Algorithm Optimization

The pseudo code of the original data processing algorithm is shown as below:

```
import pysam
for sample in dataset:
        #Fetch the reads covering the candidate variant site
        for read in pysam.fetch(sample.chr, sample.pos, sample.pos + 1):
                process_a_read(read) #Get the data
        save_a_sample() #save the data to the disk
```

In the above algorithm, the programme calls `pysam.fetch()` for each sample to fetch the reads covering the candidate variant site, get the data for each read, and save the data to the hard disk every time the processing for a sample is finished. The code underperforms for two reasons. First, the function `pysam.fetch()` takes ~0.6s to access the data regardless of the number of reads fetched in one call and thus calling the function for every sample is an unnecessary waste. Second, accessing the hard disk takes a relatively long time of several milliseconds, but the speed of writing to a hard disk can be as fast as ~100MB/s; therefore, instead of accessing and writing to the disk for each sample, we can write a much larger amount of data to the disk at one go to speed up the script.

For convenience, the processed data for each 100K samples are organized into a single .npy file. To accelerate the programme, two modifications has been made to the original algorithm. First, instead of calling `pysam.fetch()` once for each sample, the improved algorithm calls `pysam.fetch()` for all 100K samples in the same file at once and manually selects the reads covering each site. Second, the improved algorithm saves the whole file to the hard disk at one go, instead of accessing and writing to the hard disk each time the processing for a sample is finished.

The pseudo code of the improved algorithm is shown as below:

```
import pysam
import queue

read_queue = queue.Queue() #use a queue to store the "online reads"
current_sample = None

for each_file:
        #Fetch all the reads covering the 100K sites in the file
        sample_start = get_sample_start()
        sample_end = get_sample_end()
        current_sample = sample_start
        #If the samples are all on the same chromosome
        if sample_start.chr == sample_start.chr:
                for read in pysam.fetch(sample_start.chr, sample_start.pos,
                sample_end.pos + 1):
                        process_a_read(read)
        else:
        #If sample_start.chr !- sample_end.chr, the samples must be from the
        end of a chromosome and the beginning of the next chromosome
                for read in pysam.fetch(sample_start.chr, sample_start.pos,):
                        process_a_read(read)
                for read in pysam.fetch(sample_end.chr, 1, sample_end.pos + 1):
                        process_a_read(read)
        save_a_file()

def process_a_read(read):
        global read_queue, current_sample
        #Read covering current site
        if read_covering_site(read, current_sample) == True:
                #put the read into read_queue
                read_queue.put(read)
        #Read "after" current site, process the current site (maybe also the
        following sites)
        elif read.start_index > current_sample.pos or read.chr >
        current_sample.chr:
                #Process the current site, save the current read for later
                processing
                process_the_current_site(read)
        #Read "before" current site, discard the read and return
        else:
                return

def process_the_current_site(next_read):
        global read_queue, current_sample
        read_queue_temp = read_queue #After calling process_a_site(),
        read_queue will be empty; use read_queue_temp to store the information
        #Do the data processing for the current site
        data_processing(current_sample, read_queue)
        current_sample = get_next_sample(current_sample)
        #The reads covering the current site may also cover the next site!
        for read in read_queue_temp:
                if read_covering_site(read, current_sample) == True:
                        read_queue.put(read)
        #If the next_read is "before" the site, directly return
        if next_read.end_index <= current_sample.pos or next_read.chr <=
        current_sample.chr:
                return
```

```
#If next_read is "after" the current site (the current site is not
covered by the next read), we already have all the reads covering the
current site
if read_covering_site(next_read, current_sample) == False:
        process_the_current_site(next_read)
#If next_read covers the current site, put next_read into read_queue
and return
else:
        read_queue.put(next_read)
        return
```

The logic for selecting the reads covering a site is a bit tricky. In the new code, two global variables are introduced, namely `current_sample` and `read_queue`, which stores the "on-line" reads. When a new read is passed to the function `process_a_read()`, there are three possibilities: if the read covers the current site, the read is put into `read_queue` and the script moves on to the next read; if the read is "before" the current site (the end base of the read proceeds the current site), the read is discarded and the script moves on to the next read; if the read is "after the current site" (the starting base of the read succeeds the current site), it implies that all the reads covering the current site have been put into `read_queue` and the script could do the data processing for the current site by calling `process_the_current_site()`. Note that the current read needs to be passed to the function as the argument `next_read` since we still need the information to process the next sample. The function `process_the_current_site()` does two jobs: first, it does the data processing for the current site by calling the function `data_processing()`; second, it checks whether the reads in `read_queue` also cover the next site (note that a read can cover many candidate variant sites). `read_queue` is emptied after the data processing, and the content of `read_queue` is stored in another queue variable `read_queue_temp`; also, `current_sample` is set to be the next sample since the data processing for the current sample is already finished. The script puts the reads covering the `current_sample` back into `read_queue` for further processing. Last, the script checks whether `next_read` covers the current site. If so, the script puts `next_read` into `read_queue` and returns; if not and `next_read` is "before" the current site, the script discards the read and returns; if `next_read` is "after" the current site, all the reads covering the current site have already been put into `read_queue` and the function calls itself recursively by `process_the_current_site(next_read)` to process the current site.

The improved algorithm has reduced the time to process a single .npy file containing 100K samples from 3hr15m to 2hr.

4.2.2 Data Compression

The large size of the dataset has made the storage of processed data files difficult for both the long time required for writing data to the hard disk and the large space required for data storage. Without the technique of data compression, a sample consumes 590KB of disk space, and all 213M samples take up a tremendous space of 125.8TB. Hence, data compression is adopted to accelerate the script, as well as saving the disk space.

In the original implementation, a 32-bit float is used to represent each datum. However, in terms of information entropy, it is an unnecessary waste; the only possible values are 0 and 1 for elements in the 4-bit vectors, as well as for the strands. Therefore, we can use a byte instead of five 32-bit floats to store a unit: the upper two bits are used to represent the four different types of nucleotides A, C, G, and T (00 - A, 01 - C, 10 - G and 11 - T), and the lower 6 bits are used to represent the base quality score, which is represented by an integer from 0-30 and could fit into 6 bits. The null nucleotide with a null base quality score is represented by a null byte in the compressed form. Note that an A with zero base quality score and a null nucleotide couldn't be distinguished in the new representation, but it doesn't matter since the nucleotides with low base quality scores will be discarded in the design of the variant-calling network. A similar technique is employed to integrate the mapping quality and strand into a single byte; the uppermost bit is used to represent the strand, and the lower 7 bits are used to store the mapping quality, which is an integer between 0 and 60.

The data compression has reduced the size of a processed sample from 590KB to 29.6KB, and the total storage place for the processed data files from 125.8TB to 6.3TB. However, this technique has its drawbacks; the compressed data files must be un-compressed before feeding into the variant-calling network, which also consumes a considerable amount of time.

4.2.3 Parallel Computation

Parallel computation is a naïve technique but could accelerate the data processing substantially. One of the servers used in this study has 24 cores, and theoretically, 24 concurrent processes could be run on the machine without significant performance drop for each process. Hence, 24 copies of the scripts were executed concurrently, with each process dealing with 1/24 of the samples. Multi-threading or multi-processing was not chosen since both needs sophisticated programming and the data processing task requires no resource sharing or communication among the processes.

With all the three optimization techniques combined, the time consumption for data processing has been reduced from more than 280 days to ~10 days.

5 Design and Implementation of the Variant-Calling Network

Based on the results from Section 3, a variant-calling network for real-world sequence alignment data has been designed and implemented. Unlike in the prototypical problem, real-world sequence alignment contains information about the mapping qualities, base qualities, and strands besides the read sequences; these pieces of information need to be integrated into the variant-calling network to enhance the accuracy of the model.

The large size of the network has introduced difficulty to the training process for the significant memory consumption and the long training time. Moreover, recall from Section 4.2.2 that a drawback of data compression is that the compressed data file needs to be un-compressed before the training; and such pre-processing also takes considerable time. Various methods regarding the optimization of the *numpy* code for data un-compression and *keras* code for network implementation are introduced to enhance the performance of the code.

Due to the limited time available for this project, the full variant-calling network has been implemented and partially optimized but no time is left for training and fine-tuning of the network. Although no training result is available at the current stage, hopefully, the work done up-to-now could provide insights for the future studies, and I would try to offer suggestions for future works.

This section is divided into four sub-sections. Section 5.1 presents the design of the variant-calling network. Section 5.2 introduces the optimization technique and result for the data un-compression stage. Section 5.3 discusses the methods for accelerating the network. Finally, Section 5.4 would present a blueprint as well as possible suggestions for future works.

5.1 Design of the Variant-Calling Network

The design of the architecture of the variant-calling network for real-life sequence alignment data is based on the architecture of the row-invariant CNN in Section 3.1. The variant-calling network differs in two major aspects from the prototypical network: first, the size of the variant-calling network is much larger than the prototypical network, which implies that optimization is required to improve its performance; and second, unlike the randomly generated data in the prototypical problem, the real-world bioinformatics data also contains additional information from read sequences including mapping quality scores, base quality scores, and strands. Therefore, a good understanding of these pieces of information is essential for integrating them into the network to enhance the accuracy of the variant-calling network.

As stated in Section 4.1, the mapping quality score is an integer from 0-60 indicating the credibility of mapping, and the base quality score is an integer from 0-30 indicating the credibility of the base read. The official specification for the Sequence Alignment Map (SAM) format defines the mapping quality score as $-10\log_{10} p_{err}$, where p_{err} denotes the probability that the mapping of the read is incorrect. A mapping quality of 0 indicates that the mapping position of the read cannot be determined at all, for instance, in the case that two or more reigns of 148 consecutive bases are exactly the same. However, in the real-life practice, the mapping quality is generated by the Short Read Alignment Programme and the representation of mapping quality scores used by each programme may be quite arbitrary; the range and distribution of mapping quality scores generated from the same data may differ from programme to programme. For example, in some representations, the distribution of the mapping quality scores is discrete (like only 8 of the values between 0 and 60 are used), but in some other presentations, the distribution is continuous. The same problem also applies to understanding the base quality scores. The value of the base quality score is first predicted by observable properties (such as light intensity) from which the base call is extracted then calibrated with software processes, and the representation may differ from technology to technology, programme to programme. Thus, an investigation into the actual data is needed to fully understand the mapping and base quality scores.

Fig 5.1-1 presents the distributions of the mapping quality score (MAPQ) and base quality score (BASEQ) from chromosome 1 in HG002 respectively. For convenience, only the reads from chromosome 1 are sampled, whose MAPQ and BASEQ distributions can be representative for the whole genome of HG002 since the whole genome is sequenced and mapped by the same set of technological tools. From Fig 5.1-1 (left), it is obtained that ~91% of the mapping quality scores are of the highest possible value 60, and ~4% of the mapping quality scores are of 0, which indicates that the position of the read cannot be determined at all. The distribution of the mapping quality score is continuous, but for most of the scores, the percentage is too low to be visible in the graph. For the base quality scores, most of the values are distributed between 27 and 29. It could be concluded that most of the base reads and mappings are of very high credibility, but there are also a small number of reads whose mapping remains totally undetermined.

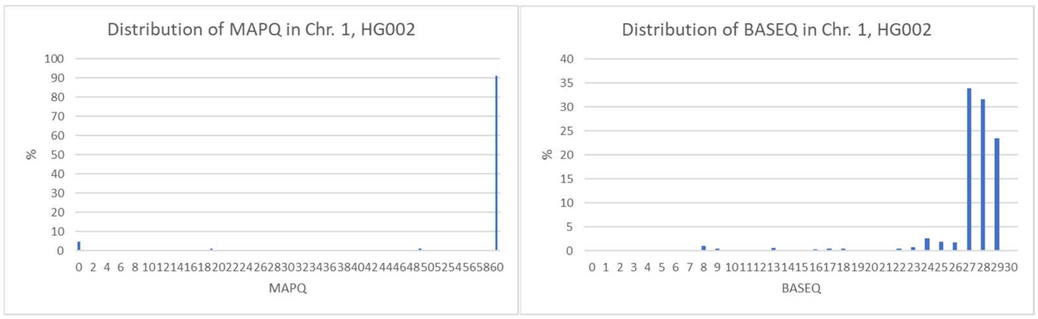

Fig 5.1-1 Distributions of MAPQ and BASEQ in Chromosome 1, HG002

(Left: MAPQ; Right: BASEQ)

In most of the variant-calling algorithms, a common practice is to dump the reads with low MAPQs and bases with low BASEQs[1,2,3]. As for *DeepVariant*, all the reads and bases with quality scores lower than 10 are discarded[1]. This study follows Google's practice. Note that for the sequence alignment data of HG002, the ranges of MAPQ and BASEQ do not equal, which implies the MAPQ and BASEQ values are not directly comparable. Thus, the raw values of BASEQ need to be multiplied with a factor of 2 to be in the same scale of MAPQ.

Although the strand carries no additional information regarding the type or credibility of the nucleotide, it also matters in the variant-calling process due to strand bias. Strand bias refers to the phenomenon that one type of DNA strand is favoured over another, which may result in incorrect evaluations in the variant-calling process. Therefore, to enhance the accuracy of variant-calling, the strand is also a crucial piece of information to consider.

The architecture of the variant-calling network is presented by Fig 5.1-2. The diagram on the left is a simplified illustration of the pre-processing of the data. The three pieces of information, namely the read sequences, the mapping quality scores, and the base quality scores need to be integrated before feeding into the network. For each nucleotide, if the minimum of MAPQ and BASEQ is less than 10, the nucleotide is discarded; otherwise, the minimum of MAPQ and BASEQ is passed through a linear mapping function and mapped to a number between 0.1 and 1, and the resulting value is multiplied with the 4-bit vector representing the nucleotide to get the processed input data. In the resulting vector, the position of the non-zero value indicates the type of the nucleotide, and the scale of the non-zero value indicates the credibility of the measurement. For example, $[\ 0.1\ 0\ 0\ 0]^T$ would represent a weak A, and $[1\ 0\ 0\ 0]^T$ a strong A. A strong A is considered more credible in the variant-calling process.

The diagram on the right demonstrates the architecture of the network, which largely resembles the network design in Section 3.1. The data, tag, and strand parts are first split; recall from Section 3.1 that the data part corresponds to the sequence alignment data around the candidate variant site and the tag part to the data on the site. Next, a row-to-row convolution is performed on the data part to retrieve the relationship between the rows; recall from Section 3.1 that the credibility of a read is correlated with whether the read is correctly mapped, which is further correlated with whether the read is the same as or similar to the other reads mapped to the same positions. Section 2 has stated that a row-to-row convolution squares the number of rows; therefore, a row-invariant average pooling is performed after the row-to-row convolution to reduce the number of rows in the processed data back to the original. After that, the processed data is concatenated with the tag and strand, and a row-invariant convolution is performed on the concatenated matrix to get the relationship among the data, tag and strand parts. Last, the processed matrix is flattened and passed through several dense layers to get the predicted genotype.

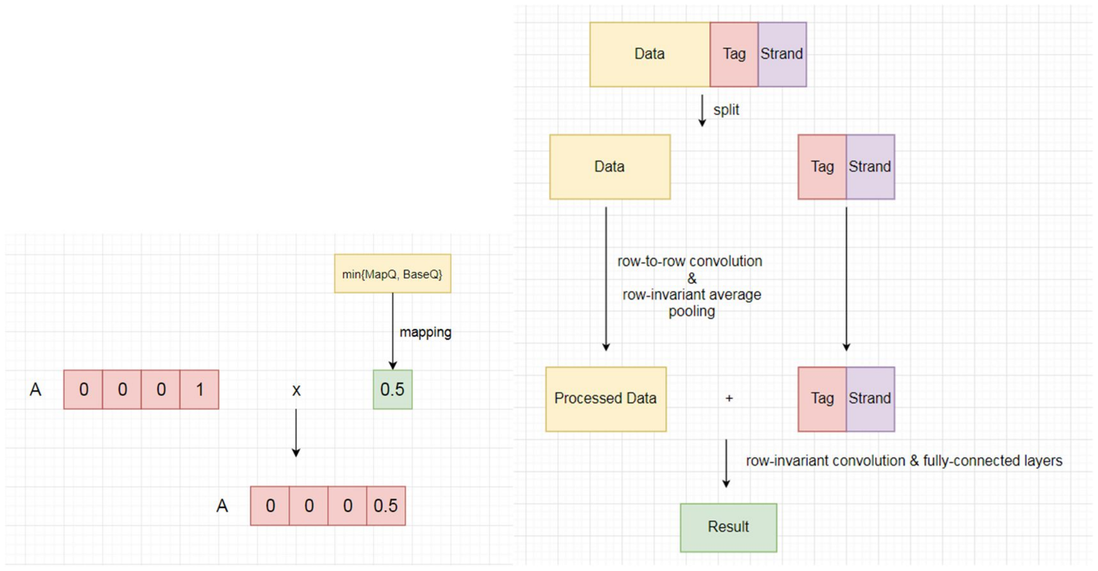

Fig 5.1-2 Architecture of the Variant-Calling Network

(Left: Pre-Processing of the Data; Right: Architecture of the Network)

Like in the prototypical problem, the script is written in *Python 3.6*, and the network is implemented with *keras* on *tensorflow* backend. The data un-compression and pre-process are done with the assistance of the *numpy* library.

5.2 Numpy Code Optimization for Bioinformatics Data Un-Compression and Pre-Process

Recall from Section 4.2.2 that a drawback of the data compression technique is that the processed data file needs un-compression before usage, which is also a time-consuming task due to the large size of the dataset. In the design of the variant-calling network, the data also needs to be pre-processed to integrate the read sequence with the mapping and the base quality scores. Both tasks were done with the assistance of *numpy* library.

The original Python code for data un-compression and pre-process is as below.

```
for i in range(num_of_sites):
        for j in range(max_read_count):
                #Get MAPQ; recall from Section 4.2.2 that MAPQ is represented
                by the lower 7 bits of the last byte in a row
                mapq = data[i][j][-1]%128
                #Discard all the reads with MAPQ <= 10
                if mapq <= 10:
```

```
                continue #skip the read
            #Get the strand; recall from Section 4.2.2 that the strand is
            represented by the uppermost bit of the last byte in a row
            strand = int(data[i][j][-1]/128)
            processed_data[i][j][-1] = strand

            for k in range(2*max_read_length - 1):
                #Get BASEQ; recall from Section 4.2.2 that BASEQ is
                represented by the 6 lower bits in a byte
                #Re-scale BASEQ by a factor of 2
                baseq = 2*data[i][j][k]%64
                #If BASEQ <= 10, discard the nucleotide
                if baseq <= 10:
                    continue
                #un-compress and process the bases
                if int(data[i][j][k]/64) == 3: #Base is "T"
                    processed_data[i][j][4*k] =
                    quality_score_scaling(mapq, baseq)
                if int(data[i][j][k]/64) == 2: #Base is "G"
                    processed_data[i][j][4*k + 1] =
                    quality_score_scaling(mapq, baseq)
                if int(data[i][j][k]/64) == 1: #Base is "C"
                    processed_data[i][j][4*k + 2] =
                    quality_score_scaling(mapq, baseq)
                if int(data[i][j][k]/64) == 0: #Base is "A"
                    processed_data[i][j][4*k + 3] =
                    quality_score_scaling(mapq, baseq)

def quality_score_scaling(mapq, baseq):
    k, b = 0.018, -0.08 #Parameters for linear scaling
    return k*min(mapq, baseq) + b
```

In the above code, the script iterates through every element of the *numpy* array to retrieve the data. The array `processed_data` stores the input data to the variant-calling network; the last elements in each row represent the strand, and all the other elements represent the processed read-sequence in units of four. The position of the element with a non-zero value indicates the type of the nucleotide, and the value the credibility of it. To enhance the performance of the code, *numpy* slicing has been adopted instead of iteration; prototypical scripts have shown that the performance of slicing largely surpasses that of iteration. The improved code is shown as below.

```
#Get MAPQ
mapq = data[:,:,-1]%128
mapq = (mapq.repeat(2*max_read_length-
1)).reshape(num_of_sites,max_read_count,2*max_read_length-1)
#Get BASEQ; scale BASEQ with a factor of 2
baseq = 2*data[:,:,:2*max_read_length - 1]%64
#Get strand
processed_data[:,:,-1] = np.floor(data[:,:,-1]/128)
#Get nucleotides; pre-process the vectors
k, b = 0.018, -0.08
```

```
processed_data[:,:,:4*(2*max_read_length-1):4] =
np.absolute((k*np.minimum(baseq, mapq)+b)*(np.sign(baseq-
10.1)+1)/2*(np.sign(mapq-
10.1)+1)/2*np.floor(data[:,:,:2*max_read_length-
1]/64)*(np.floor(data[:,:,:2*max_read_length-1]/64)-
1)*(np.floor(data[:,:,:2*max_read_length-1]/64)-2))
#(k*np.minimum(baseq, mapq)+b): equivalent to quality_score_scaling()
in the above code
#(np.sign(baseq-10.1)+1)/2: check if baseq > 10, otherwise discard the
nucleotide; I used 10.1 instead of 10 to eliminate the case when
np.sign() = 0
#(np.sign(mapq-10.1)+1)/2: check if mapq > 10, otherwise discard the
nucleotide
#np.floor(data[:,:,:2*max_read_length-
1]/64)*(np.floor(data[:,:,:2*max_read_length-1]/64)-
1)*(np.floor(data[:,:,:2*max_read_length-1]/64)-2):
#the upper two bits of data[:,:,:2*max_read_length-1] were used to
represent the nucleotides
# 0-A, 1-C, 2-G, 3-T
#the terms are to check whether the two upper bits represent zero (!=
1, 2 and 3)
processed_data[:,:,1:4*(2*max_read_length-1):4] =
np.absolute((k*np.minimum(mapq, baseq)+b)*(np.sign(baseq-
10.1)+1)/2*(np.sign(mapq-
10.1)+1)/2*np.floor(data[:,:,:2*max_read_length-
1]/64)*(np.floor(data[:,:,:2*max_read_length-1]/64)-
1)*(np.floor(data[:,:,:2*max_read_length-1]/64)-3))
processed_data[:,:,2:4*(2*max_read_length-1):4] =
np.absolute((k*np.minimum(mapq, baseq)+b)*(np.sign(baseq-
10.1)+1)/2*(np.sign(mapq-
10.1)+1)/2*np.floor(data[:,:,:2*max_read_length-
1]/64)*(np.floor(data[:,:,:2*max_read_length-1]/64)-
2)*(np.floor(data[:,:,:2*max_read_length-1]/64)-3))
processed_data[:,:,3:4*(2*max_read_length-1):4] =
np.absolute((k*np.minimum(mapq, baseq)+b)*(np.sign(baseq-
10.1)+1)/2*(np.sign(mapq-
10.1)+1)/2*(np.floor(data[:,:,:2*max_read_length-1]/64)-
1)*(np.floor(data[:,:,:2*max_read_length-1]/64)-
2)*(np.floor(data[:,:,:2*max_read_length-1]/64)-3))
```

Since there is no appropriate *numpy* built-in functions to handle the case statement (`np.where()` only applies to one-dimensional arrays), I had to use above-shown clumsy method to retrieve the type of the nucleotide and do the pre-processing. For an element of a unit in `processed_data`, the value is only non-zero when the type of the nucleotide is the type corresponding to the position of the unit. In the four expressions used to determine the value of an element, the last three terms are used to check the type of the nucleotide; for instance, the term `np.floor(data[:,:,:2*max_read_length-1]/64` checks if the nucleotide is an A, and if the nucleotide is indeed an A, the corresponding element will be set to zero since the element would only be of non-zero value if the nucleotide is a T. If the nucleotide is not of type A, C, and G, it could be determined that the nucleotide is a T or a null nucleotide, and the element may be of a non-zero value. The second and third terms in each expression checks if the MAPQ or BASEQ is less than or equal to 10 and discard the nucleotide accordingly. The first term does the linear scaling of the minimum of MAPQ and BASEQ, which performs the same function as `quality_score_scaling()` in the original script.

With the above technique, the time consumption of un-compressing and pre-processing a data file could be reduced from 8 hrs to 0.5 hrs. Note that the technique may not be the best solution due to my limited time available for dealing with this issue, but the performance is sufficient at the current stage.

5.3 Keras Code Optimization for Large Neural Networks

The large scale of the variant-calling network has introduced great difficulty to the project for both its large GPU memory consumption and long training time. With three optimizations, namely reducing the filter size for row-to-row convolution, reducing the number of reads, and moving part of the workload from GPU to CPU, the network size has been successfully reduced to 1/5 its original, and the training time for an epoch of 90K training samples has dropped from ~23h to ~1h.

Though the afore-mentioned techniques for optimizing the network has achieved promising results, the study on variant-calling network optimization remains uncomplete. Currently, to train the network on one data file with 100K samples (90K for training and 10K for validation) still takes ~4 days, not to mention there are 2130 data files in total. Though using a subset of the data may be sufficient for the network to converge, to accelerate the network training process would undeniably benefit the future works on fine-tuning and finalizing the network.

This section is structured as follows. Section 5.3.1 discusses the influence of reducing the filter size for the row-to-row convolution on the network performance. Section 5.3.2 presents the method of finding the optimal number of rows for the input matrix. Section 5.3.3 discusses how moving part of the workload from GPU to CPU affects the network performance.

More insights on and suggestions for the future works will be presented in Section 5.4.

5.3.1 Adjustment on the Filter Size for Row-to-Row Convolution

One of the optimizing techniques is to reduce the filter size in row-to-row convolution. Recall from Section 2 that in the process of row-to-row convolution, the matrix size is squared and then doubled, which implies that the row-to-row convolution is the bottleneck for the GPU memory consumption for the variant-calling network. In Appendix II, the memory consumption for each layer in the original variant-calling network design is shown, from which it could be seen that the row-to-row convolution consumes over 90% of the GPU memory consumed by the variant-calling network. In the implementation of the row-to-row convolution, to realize the effect of using a filter size of a, copying the original matrix for a times is required. Therefore, both the size of the input matrix and the filter size affects the memory consumption of the row-to-row convolution layers. Therefore, there are two major ways to reduce the memory consumption of the variant, namely to reduce the size of the filter and to reduce the size of the matrix; the former method will be discussed in this section and the latter in Section 5.3.2.

Recall from Section 3.1 that in the prototypical network, a filter size of 3 has been adopted. To verify that a reduce of the filter size would not adversely affect the accuracy significantly, a new network has been designed and implemented to solve the same prototypical variant-calling problem. With the architecture and all other parameters being the same, the filter size for the row-to-row convolution has been reduced from 3 to 1. With 2M data and 5 hours of training, the new network could achieve an MSE of 0.0169 comparing to 0.0005 of the original network and requires a number of trainable parameters of 30,182 compared to that of 30,662 in the original network. Therefore, reducing the size of the filter could decrease the accuracy of the network, but such drop is not significant; and by adjusting the size of the filter, some accuracy may be traded for performance.

Reducing the filter size from 3 to 1 could theoretically reduce the GPU memory consumption of the network to 1/3 its original. Since the memory consumption of the network negatively correlates to the maximum batch size for network training, and the batch size linearly correlates to the training speed of the network, the optimization could theoretically speed up the training by a factor of 3.

5.3.2 Adjustment on the Size of the Input Matrix

Another approach to reducing the GPU memory consumption of the network is to reduce the size of the input matrix. In Section 4, the number of rows in the input matrix has been set to 100, yet it is in question whether it is the optimal dimension. With the assistance of a script that generates the statistics about the number of reads covering each of the candidate variant sites, it was obtained that the average number of reads covering each site is 39, with a standard deviation of 10. Hence, by reducing the number of rows in the input matrix to 60, for >95% of the sites, all the reads covering could be included in the matrix. Given that the memory required for row-to-row convolution is correlated to the square of the number of rows, such a change could theoretically reduce the memory consumption to 36% of its original. The number of rows in the input matrix could be further adjusted to decrease the size or to increase the accuracy of the network.

Moreover, only using part of the read sequences (decreasing the number of columns) can also decrease the size but may c the accuracy of the variant-calling network. Nevertheless, due to the limited time available for this study, no experiment has been done on this proposed optimization.

The above two methods have reduced the GPU memory consumption of the network from 1.338GB to 0.262GB, which is about 1/5 of the original.

5.3.3 Keras Code Optimization

The original code for the variant-calling network is shown as below:

```
inputs = Input(shape = (60, 1181), dtype = "float32")
#Separate the inputs into data and tag & strand
tag = Lambda(lambda x: x[:, :, 588:592])(inputs)
strand = Lambda(lambda x: x[:, :, 1180:])(inputs)
tag_and_strand = concatenate([tag, strand])
data_1 = Lambda(lambda x: x[:, :, :588])(inputs)
data_2 = Lambda(lambda x: x[:, :, 592:1180])(inputs)
data = concatenate([data_1, data_2])
#row-to-row convolution on data
data = Reshape((60, 294, 4))(data)
data_processed = Flatten()(data_processed)
data_processed = RepeatVector(60)(data_processed)
data_processed = Reshape((60, 60, 294, 4, 1))(data_processed)
data_transpose = Permute((2, 1, 3, 4, 5))(data_processed)
data_combined = concatenate([data_processed, data_transpose])
data_combined = Reshape((1058400, 8))(data_combined)
data_combined = Conv1D(30, 1, padding='valid',
activation='selu')(data_combined)
data_combined = Conv1D(8, 1, padding='valid',
activation='selu')(data_combined)
#row-invariant average pooling
```

```
data_combined = AveragePooling1D(pool_size=17520, strides=None,
padding='valid')(data_combined)
#concatenate data and tag&strand and perform row-invariant convolution
input_combined = concatenate([tag_and_strand, data_combined])
input_combined = Conv1D(13, 1, padding = 'valid', activation =
'selu')(input_combined)
input_combined = Conv1D(8, 1, padding = 'valid', activation =
'selu')(input_combined)
input_combined = Flatten()(input_combined)
input_combined = Dense(500, activation = 'selu')(input_combined)
input_combined = Dense(80, activation = 'selu')(input_combined)
input_combined = Dropout(0.2)(input_combined)
predictions = Dense(10, activation = 'softmax')(input_combined)
```

The first few lines of the code have been highlighted in yellow. Note that the operations performed by the highlighted lines, including slicing, permutation, concatenation and reshaping, are sequential in nature. Since CPU is better at performing sequential operations than GPU, moving these codes to CPU may accelerate the script. Nevertheless, since the lines in red consumes a considerable amount of memory after the `RepeatVector()` operation, with the verification of experimental scripts, *numpy*'s performance on these large-scale arrays is actually worse than that of the original implementation. Therefore, after some experiments, it was found that moving the code in blue to CPU gives the best performance. The improved code is shown as below:

```
temp = np.copy(processed_data[:, :, 588:592])
processed_data[:, :, 588:592] = processed_data[:, :, 1176:1180]
processed_data[:, :, 1176:1180] = temp
inputs = Input(shape = (60, 1181), dtype = "float32")
#Separate the inputs into data and tag & strand
tag_and_strand = Lambda(lambda x: x[:, :, 1176:])(inputs)
data = Lambda(lambda x: x[:, :, :1176])(inputs)
#Row-to-row convolution on data
data_processed = Flatten()(data)
data_processed = RepeatVector(60)(data_processed)
data_processed = Reshape((60, 60, 294, 4, 1))(data_processed)
data_transpose = Permute((2, 1, 3, 4, 5))(data_processed)
data_combined = concatenate([data_processed, data_transpose])
data_combined = Reshape((1058400, 8))(data_combined)
#Rest of the code…
```

In the improved code, before the input passed to the variant-calling network, the data unit at the candidate variant site and the data unit at the last base in the matrix are first swapped with *numpy* array operations, so that fewer number of sequential operations need to be performed by *keras* layers. This optimization has decreased the time required for training the network on a dataset of 90K training samples for one epoch ~1.5hrs to ~1hrs.

Combining the three optimization techniques, the GPU memory consumption of the variant-calling network has been reduced to 1/5 of its original, and the time required to train the network on one processed data file (with 90K samples for training and 10K for validation) from ~23hrs to ~1hrs. The optimizing result seems promising at the current moment, but further optimization may still be necessary for boosting the performance and improving the accuracy of the network. Suggested methods for further optimization will discussed in the next sub-section.

5.4 Future Works

Due to the limited time available for this project, the training, fine-tuning, and finalization of the variant-calling network are not included in the scope of this study. This section aims to provide insights into, as well as offering possible suggestions for the future works.

As stated in Section 5.3, although the current optimizations have achieved promising results, further optimizations are still needed to enhance the performance of the script for the convenience of future studies. Based on the existing work, a few suggestions are offered for the optimization of the variant-calling network.

First, recall from Section 5.3.1 and 5.3.2 that the current bottleneck of the network performance is the row-to-row convolution, for performing which the size of the network layer needs to be squared and doubled. In the current implementation, the matrix also needs to be duplicated for a times for a filter size of a. However, these duplications are only for the ease of implementation but not essential for performing the operation; with an improved software implementation of the row-to-row convolution, say writing a customized *keras* layer for it, such duplications could be avoided, and the size of the network could be largely reduced. With a significantly reduced GPU memory consumption, the batch size used in the network training could be increased, which could substantially accelerate the training. Such optimization only changes the implementation method, but not the function or scale of the row-to-row convolution operation, and thus would not reduce the accuracy of the network. Moreover, recall from Section 5.3.1 that the filter size could affect the accuracy of the variant-calling network, though such influence is not substantial. Therefore, the above optimization will also enable the filter size to be enlarged without significantly increasing the network size, so that the accuracy of the network could be enhanced.

Second, the accuracy may be partially traded for a higher training speed. The optimizations done in Section 5.3.1 and 5.3.2, namely adjusting the filter size and number of rows in the input matrix, constitute examples of such approaches. Another proposed technique is to use fewer bits to represent the MAPQ and BASEQ, given that the majority of MAPQs and BASEQs are of or close to the highest possible score.

Third, part of the code may be re-written in a lower-level language like C/C++ for acceleration.

Fourth, a deeper understanding of the strand's influence on the variant-calling process may inspire for an enhanced network design.

The above-proposed techniques may further accelerate the network training and benefit the further training and fine-tuning.

6 Conclusion

Google's *DeepVariant* has achieved promising results in applying convolutional neural networks to variant-calling problems. However, with its relatively large number of trainable parameters, the application of *DeepVariant* is currently limited to fields where sufficient amounts of data are available. By exploiting the row-invariant property in sequence alignment data, a row-invariant convolutional neural network can be designed and built, which is much more compact than a convolutional neural network; therefore, the row-invariant network requires a much smaller amount of training data and shorter training time than the state-of-the-art network.

The aim of this project is to design and build a row-invariant convolutional neural network for variant-calling problems. A prototypical variant-calling problem has been designed to validate the row-variant CNN's superior performance over the CNN's on treating data with the row-invariant property, and the variant-calling network for real-world bioinformatics data has been designed, implemented and optimized. The current optimization has given promising results, yet future work on the network optimization is necessary for the network to achieve optimal performance and accuracy. Due to the limited time, the fine-tuning and finalization of the variant-calling network are not included in the scope of the project; nevertheless, hopefully, the current work could provide insights into the direction of future work. Given appropriate optimization and tuning, the variant-calling network may be expected to beat *DeepVariant*'s performance.

The last remark is that the application of row-invariant CNN's may be potentially expanded to other bioinformatics process involving sequence alignment data, or other datasets with the row-invariant property.

Appendix I: Important Source Codes

See the GitHub Link: https://github.com/aliciaxue/sdp_source_codes

Appendix II: Memory Consumption of the Variant-Calling Network (Original)

See the GitHub Link: https://github.com/aliciaxue/sdp_source_codes

References

1. Poplin, R. Newburger, D. Dijamco, J. Nguyen, D. Loy, D. Gross, S.S. McLean, C.Y. DePristo, M.A. (2016). “Creating a universal SNP and small indel variant caller with deep neural networks.” Retrieved from https://research.google.com/pubs/pub46409.html.

2. Nielson, R. Paul, J.S. Albrechtsen, A. Song, Y.S. (2011). “Genotype and SNP calling from next-generation sequencing data.” Nature Reviews Genetics, vol. 12, pp. 443–451.

3. DePristo, M.A. Banks, E. Poplin, R. Garimella, K.V. Maguire, J.R. et al. (2011). “A framework for variation discovery and genotyping using next-generation DNA sequencing data”. Nature Genetics, vol. 43, iss. 5, pp. 491-498.