**Row-invariant Convolutional Neural Network with Applications to Bioinformatics**

Yafei Xue (UID: 3035142405)

***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 can be built, which thus requires a smaller amount of training data and a 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**

**1 Introduction**

Variant-calling is an important step in many forms of bioinformatics data analysis. 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 approaches1. However, with over 20M parameters, the application of *DeepVariant* is currently limited to the fields where sufficient data is available1. 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 can thus largely reduce the required amount of training data and 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, the current work could bring valuable insights for future works.

The paper is structured as follows. Section 2 introduces the theoretical basis of designing a row-invariant convolutional neural network. Section 3 describes the design, implementation, and performance of prototypical networks. Section 4 explains the data processing process, including remarks on Python code optimization on large-scale bioinformatics data. Section 5 presents 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.

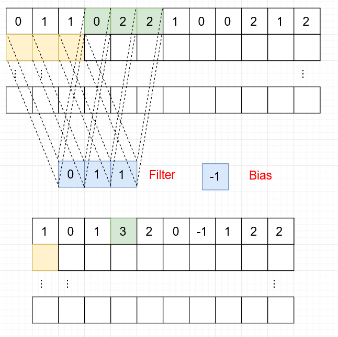
**2 Theoretical Basis of Designing a Row-invariant Convolutional Neural Network**

The structure of a row-invariant convolutional neural network derives from that of a convolutional neural network (CNN). With successful applications in image processing, CNN’s take advantage of the 2D structure of the input images; in a convolutional network layer, the hidden units are only connected to a small area of the input and convolve with the whole image to produce the output, thus requiring far fewer parameters than a fully-connected neural network. 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.

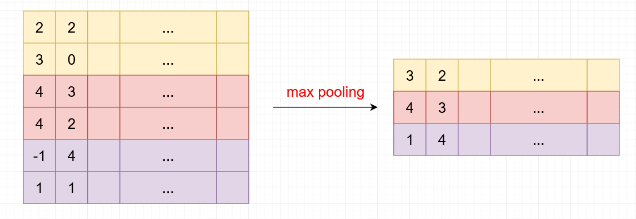
Unlike images, sequence alignment data has the property of being row-invariant: that is, the rows in a sequence alignment data are interchangeable. The design of a row-invariant CNN aims to exploit this row-invariant property by introducing 1D convolution and 1D pooling to further reduce the number of 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 Figure 2-1. The input matrix is of the size , where denotes the number of columns and the number of rows. The row-invariant convolutional layer takes a 1D kernel with the length , which convolves with each row respectively to produce a feature map with the size . Replacing a 2D kernel with a 1D kernel, a row-invariant convolutional layer requires a much fewer number of parameters compared to a convolutional layer.

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.

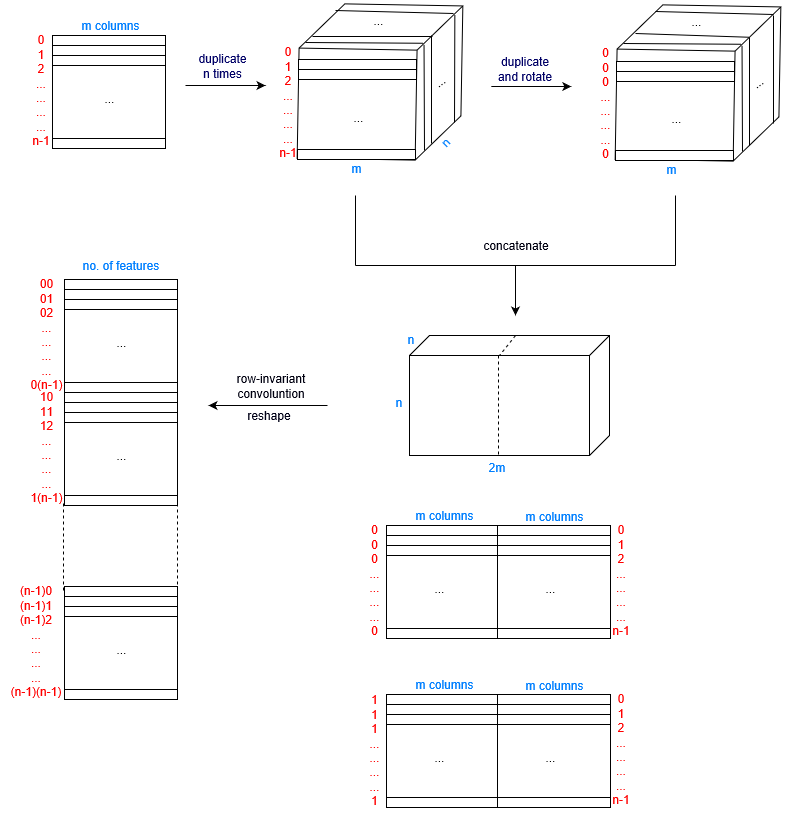


**Fig 2-1** Illustration of Row-Invariant Convolution

****

**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 ; the matrix is duplicated for times, producing a 3D matrix with the dimensions . The 3D matrix is further duplicated, rotated, and concatenated with the original 3D matrix. The concatenated matrix is reshaped into a 2D matrix with the dimensions , 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.



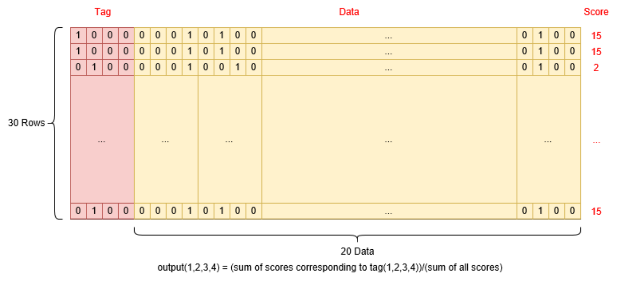
**Fig 2-3** Illustration of Row-to-Row Convolution

**3 Design, Implementation, and Performance of the Prototype**

The prototype is a simplified version of the variant-calling problem and is designed to verify the performance of a row-invariant CNN. While the real-world sequence alignment data contains four pieces of information relevant to variant-calling, namely nucleotide sequences, base and mapping quality scores and 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. 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 the similarity between a row and other rows.

Following the above simplifications and assumptions, the prototype problem is designed as follows, with its illustration presented by Fig 3-1:

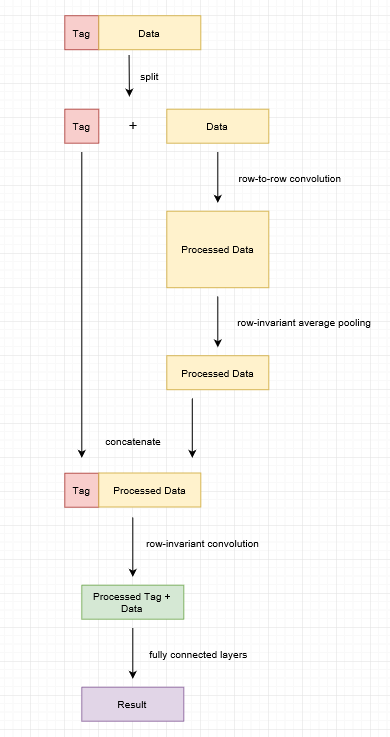
*Consider a matrix with 30 rows. Each row has a tag and 20 data, and each tag or datum is represented by a 4-bit, one-hot or zero-hot vector. Consider the number of rows with the same data parts (all 20 pairs of vectors being equal) and assign each row with a score. For example, for the first row, if 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.*



**Fig 3-1** Illustration of the Prototypical Problem

In the prototype problem, the tag part corresponds to the nucleotides at the candidate variant site and the data part 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 each type of nucleotide at the site, which is estimated from the probabilities that the rows are correctly mapped. The score of each row, which represents the extent that the row is the same as other rows, 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 3-2. 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 the original while preserving the feature information for the convenience of later operations. Finally, since the relationship between the scores and corresponding tags is essential for calculating the likelihoods, 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 3-2** Architecture of the Prototype 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 CPUs and GPUs, 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 .

Table 3-3 compares the performance of a row-invariant CNN and a CNN on the prototype problem. A filter size of 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 3-3** Comparison between a CNN and a Row-Invariant CNN on the Prototypical Problem

**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 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 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.

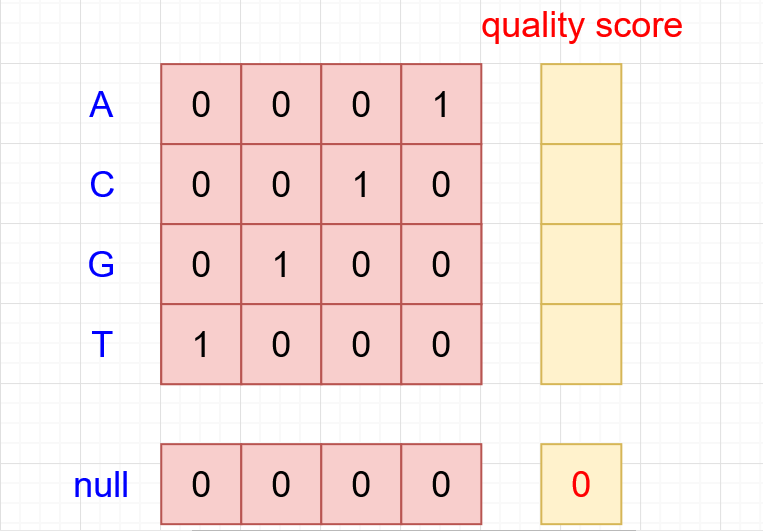
2) The base quality score sequence, represented by an array of base quality scores corresponding to each base in the read sequence. The base quality score is an integer from 0-30, indicating the likelihood that the base read is correct.

3) The mapping quality score, represented by an integer from 0-60, indicating the likelihood that the mapping of the read is correct.

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

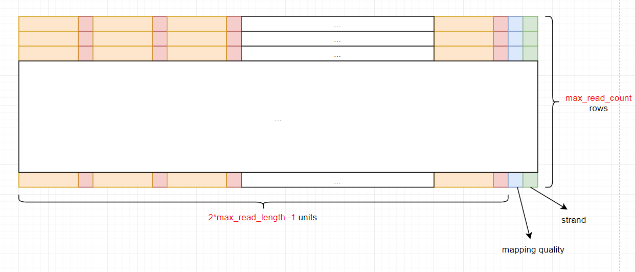
As in Section 3, in the processed data, each base is represented by a 4-bit, one-hot vector corresponding to the type of the nucleotide A, C, G, or T, and a null nucleotide is represented by a 4-bit, zero-hot vector. A null nucleotide may indicate a deletion, or a base not covered by the read. The correspondence between the nucleotide and the 4-bit vector is illustrated 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 consisted of (2\*max\_read\_length - 1) units, where max\_read\_length gives the length of a read, 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. In each row, there are two additional pieces of information; an integer representing the mapping quality score, and the other one the strand of the read, represented by an integer being 0 or 1. For each candidate variant site, if the number of reads covering the site exceeds the number of rows in the matrix, only the reads with maximum mapping qualities will be included in the matrix.

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



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

**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.

a) Algorithm Optimization

In the original implementation, the script iterates over all the candidate variant sites and calls the function pysam.fetch() for each site to fetch the corresponding reads; and the data is saved to the disk each time the processing for a site 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, since accessing the hard disk takes a long time comparing to the speed of writing data to the disk, it’s preferable to write a large amount of data to the disk at one go. Therefore, to accelerate the script, two modifications have been adopted. First, instead of calling pysam.fetch() once for each sample, the improved algorithm calls pysam.fetch() for a large number of consecutive samples 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 improved algorithm has reduced the time to process a single .npy file containing 100K samples from 3hr15m to 2hr.

b) 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 reducing the disk space consumption.

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, a byte instead of five 32-bit floats can be used 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 an integer between 0 and 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.

c) Parallel Computation

Parallel computation is a naïve technique but could accelerate the data processing substantially. Multiple concurrent processes were run on a server with multiple cores without significant performance drop for each of the processes, and each process could deal with a subset of the data.

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. Various methods regarding the optimization of the *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.

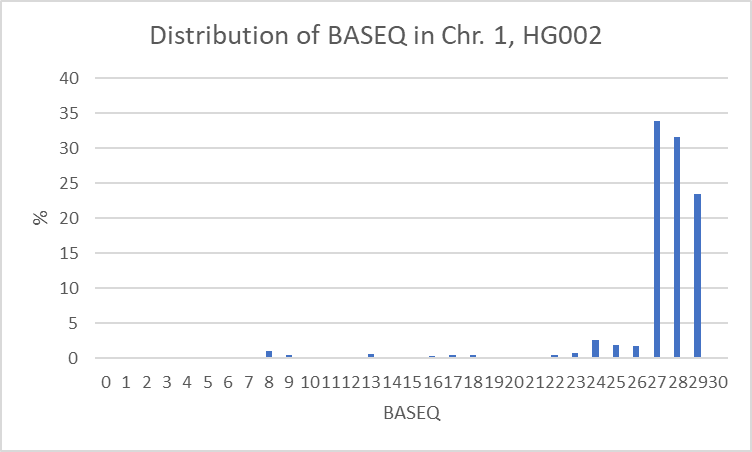
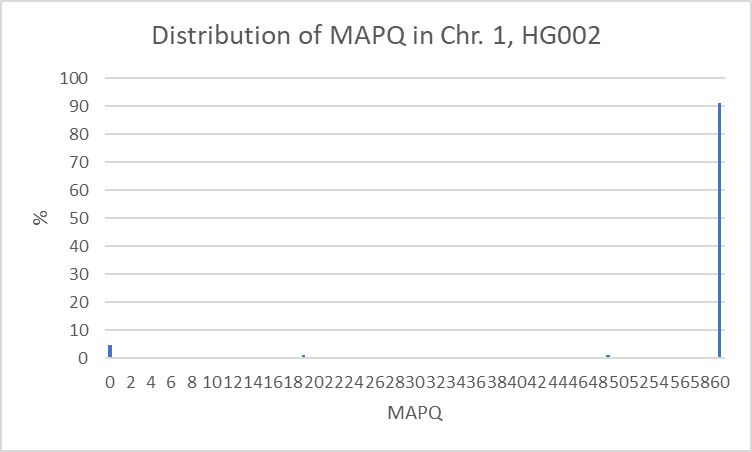
This section is structured as follows. Section 5.1 presents the design and implementation of the variant-calling network. Section 5.2 introduces the optimization technique for accelerating the network. Finally, Section 5.3 would present a blueprint for future works.

**5.1 Design of the Variant-Calling Network**

The design of the variant-calling network for real-life sequence alignment data is based on the architecture of the row-invariant CNN in Section 3. 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 relevant to variant-calling.

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 , where denotes the probability that the mapping of the read is incorrect. 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. Thus, an investigation into the actual data is essential 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.



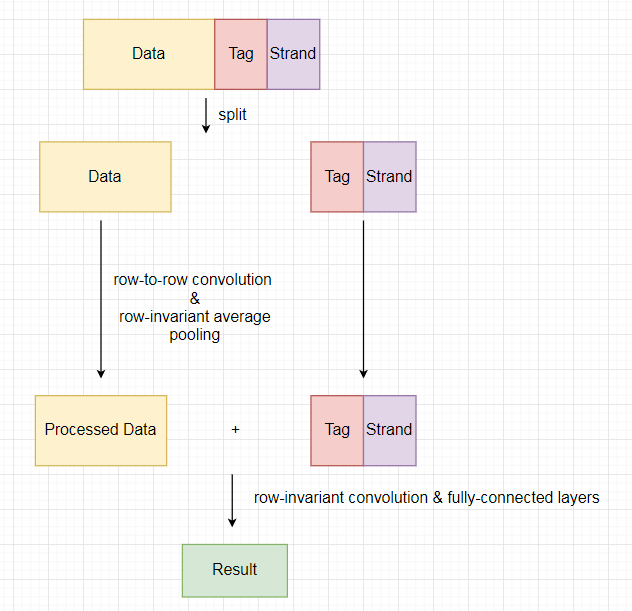
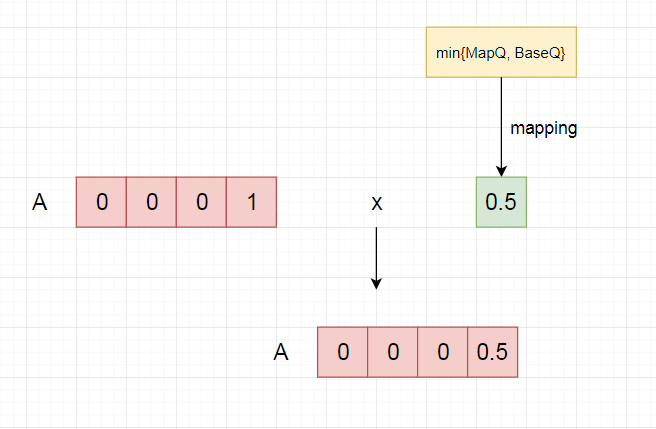
**Fig 5.1-1** Distributions of MAPQ and BASEQ in Chromosome 1, HG002 (Left: MAPQ; Right: BASEQ)

In most of the variant-calling algorithms, a common practice is to dump the reads with low MAPQ’s and bases with low BASEQ’s1,2,3. As for *DeepVariant*, all the reads and bases with quality scores lower than 10 are discarded1. This study follows Google’s practice. Also, due to the difference in ranges of MAPQ and BASEQ, the raw value of BASEQ’s was multiplied by a factor of two to be directly comparable with MAPQ’s.

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 at the top 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, would represent a weak A, and a strong A. A strong base is considered more credible in the variant-calling process.

The diagram at the bottom demonstrates the architecture of the network, which largely resembles the network design in Section 3. The data, tag, and strand parts are first split; recall from Section 3 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; also recall from Section 3 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

(Top: Pre-Processing of the Data; Bottom: 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 Keras Code Optimization for Network Training**

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 training 90K samples for an epoch 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 100K samples (90K for training and 10K for validation) still takes ~4 days, not to mention there are 213M samples in total. Though 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.

a) 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 , copying the original matrix for 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.

Recall from Section 3 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.

b) Adjustment on the Size of the Input Matrix

Another approach to reducing the GPU memory consumption of the network is to determine the optimal size of the input matrix. 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 from 100 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 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.

c) Keras Code Optimization

The variant-calling network is realized in *keras* on *tensorflow* backend, which distributes the computational workload to GPUs. Nevertheless, some of the operations in the variant-calling network, including slicing, permutation, concatenation, and reshaping, are sequential in nature. Since CPU has a superior performance at sequential operations compared to GPU, moving relevant codes to CPU can accelerate the script. Such 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 100K samples (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 training speed and improving the accuracy of the network. Suggested methods for further optimization will be discussed in the next sub-section.

**5.3 Proposed Methods for Future Optimization**

Based on the existing work, a few suggestions are offered for the future optimization of the variant-calling network.

First, recall from Section 5.2 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 times for a filter size of . 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, 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 5.2(a) 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 5.2(a) and (b), 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.

**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.

**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 express my gratitude to 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.

**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.

**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>