Comparing two genome sequences using a Recurrent Neural Network

Dorothy Cheung, Department of Computer Science and Engineering, *University of Nevada Reno, Reno USA*

Abstract— The research area of genome sequencing is prominent in bioinformatics and computer science. Although there are methods on genome sequencing, the process requires continuous improvement on speed, reliability, and cost. The sequences obtained for assembly are not truly accurate due to machine and human handling mistakes. We need to consider a methodology which allows for errors when the sequences are compared for assembly. A Neural Network is designed to work with uncertainty. This paper describes the possibility of using a recurrent Neural Network to determine the exact similarity between two genomic sequences. The preliminary results show that it can be promising.

***Index Terms*—Bioinformatics, Sequence Assembly, Recurrent Neural Network**

I. Introduction

Understanding the formation of living cells is an important concept of Genomic, allowing scientists to make necessary modifications on cells affected by deadly viruses, for example. A genome represents a set of instructions encoded in DNA sequences. DNA or Deoxyribonucleic acid is a nucleic acid that contains the genetic instructions necessary for constructing cells in living organisms. It is composed of four nucleotides or base pairs: A, C, G, and T.

The entire genome sequence may contain several thousands to millions of base pairs, but researchers are only able to decode about 600 to 700 base pairs at one time from chemical reactions [2]. To overcome this limitation, scientists have developed a technique called "shotgun sequencing." The technique involves breaking up the original sequence into smaller sets of raw sequences, sequencing them into readable form, and putting them together using an assembler. DNA or genome sequencing is the process of determining the order of these broken up base pairs.

The genome sequencing process using the "shotgun sequencing" technique can be divided up into three critical steps: reading, assembling, and finishing, as shown in figure 1 [1].

Reading sequences is the process of feeding raw sequences into a sequence reader and obtaining sequences stored in chromatogram files, which are then transformed into readable sequences of characters A, C, G, and T that represent the nucleotides. Multiple comparisons of shotgun sequences (reads) are needed to ensure that all regions are covered since the genome sequence is too large to be processed in its entirety and raw sequences are extracted in random.

Assembling is the process of putting together the sequences by identifying the similar regions between them. The matching regions are also known as "Contigs." Errors in obtaining the sequences contribute to the difficulty of similarity matching. The imperfection can cause insertions or deletions in the sequences, causing most applications to reject matching sequences.

Finishing is the manual process of fiddling with the assembled sequence to fill in any gaps in the sequence. The efficiency is undermined by the assembling process.

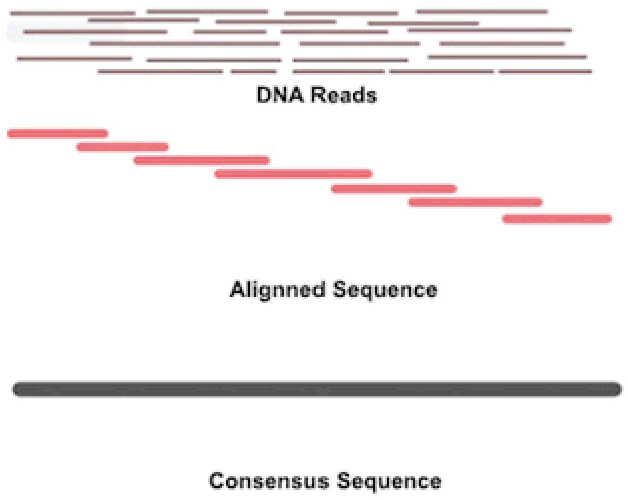

Fig. 1. The genome sequencing process [1].

There are several problems with genome sequencing. The sequence read from a machine can contain errors as shown in figure 2. Sequence 1 and 2 are approximately similar sequences, but sequence 1 has an insertion of the character 'G' and the same character is deleted in sequence 2. The data retrieved from the laboratory can be of low quality. Repeats or repetitive sub-sequences further complicate the process because sub-sequences can either be combined or placed at multiple regions as illustrated in figure 3.

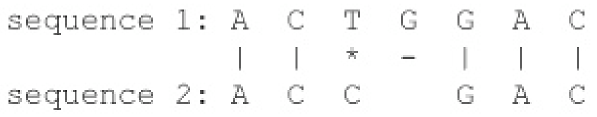

Fig. 2. Problems with genome sequencing. Sequence 1 and 2 are approximately similar sequences. The symbol “-” indicates a missing character from gene deletion or insertion. The symbol “*” indicates a change in the sequence.

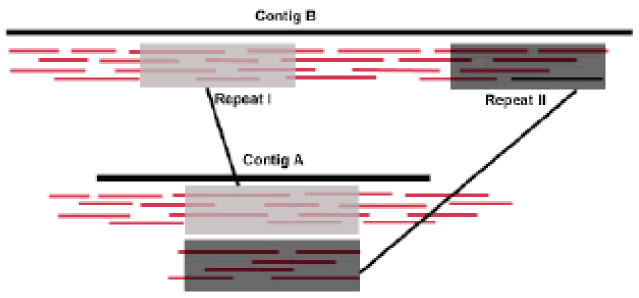

Fig. 3. Repeats in Contigs. Region I and II can be combined in Contig A or assembled at different regions in Contig B [1].

Therefore, the goal of genome sequencing is not to put together the exact matches of sequences, but to assemble a sequence with some degree of certainty. Although there are existing tools for sequence analysis, most of these applications have low tolerance on errors [2]. Some assemblers account for the number of matches to determine if there are repeats since the number of times a sequence can appear is restricted by the number of “reads.”

There are many algorithms that can compare two genome sequences including different variations of dynamic programming, which solves a problem by dividing the solutions into smaller problems [8]. The algorithm widely mentioned is the Smith-Waterman algorithm, which accounts for deletions and insertions of arbitrary length [5]. It has been replaced by the BLAST (Basic Local Alignment Search Tool) algorithm, which is faster but less accurate than Smith-Waterman. A BLAST search allows researchers to identify sequences from the sequence database that resemble the query sequence. It also enables users to select a particular genome to “BLAST” against [6].

Other techniques include suffix tree [11], KMS algorithm [10], fuzzy logic [1], and greedy approaches that can be faster than dynamic programming [9].

This paper proposes using a Neuroevolution technique that determines the exact similarity between two sequences of the same length. A recurrent Neural Network is needed in this application because the current outcome is affected by the previous inputs. Training with a Genetic Algorithm (GA), developed by Goldberg and Holland, is preferred over Backpropagation because Backpropagation through time requires complex calculations. The following sections describe the procedure and summarize preliminary experimental results that support the Neuroevolution technique.

II. Methodology

A. Data sets

The sequential input into the Neural Network consists of the “localist encoding” of two characters from the two sequences in comparison. This encoding scheme uses four bits and activates a bit to identify each of the four characters {{G: 1000}, {T:0100}, {C:0010}, {A:0001}}.

The input data is generated by splicing the original DNA sequence from the Homo sapiens (humans) calcium channel consisting of 5909 base pairs. The gene is “CACNA1S”, locus_tag = “BC133671” and the GeneID is “779” obtained from GenBank [4]. The original DNA sequence needs to be thousands of characters long to create a diversified data set. The number of base pairs in each split may vary, but each split should contain at least 40 base pairs. Otherwise, similar classification can be invalid due to errors introduced in the process of genome sequencing.

Four data sets are generated for this experiment: training, validation, tuning test, and “hold out” test. Each data set contains the same number of similar and dissimilar corpuses. A corpus consists of two sequences to be compared. Due to time constraints in running the experiment, there are 4000 corpuses for the training set and 1000 for the other sets. All splits are 40 characters long.

B. Recurrent Neural Network

Determining similarity between two sequences takes into account the previous activation of the hidden layer. As shown in figure 4, a simple recurrent Neural Network is designed to include an additional context layer, where the activation of the hidden units from the previous feed-forward process is stored [3]. Four input nodes are used to represent each bit of an encoded character. A total of eight input nodes are needed to account for both characters from the two sequences being compared. Ten hidden and context nodes are used in this experiment because of time constraints in training the recurrent Neural Network. The hyperbolic tangent function is used to squash the results within the range [-1, 1]. The output is expected to be 0.9 if all the characters that the network has seen thus far are the same and -0.9 otherwise. In addition, a bias unit is added to both the input and hidden layers. The concept is illustrated in figure 5.

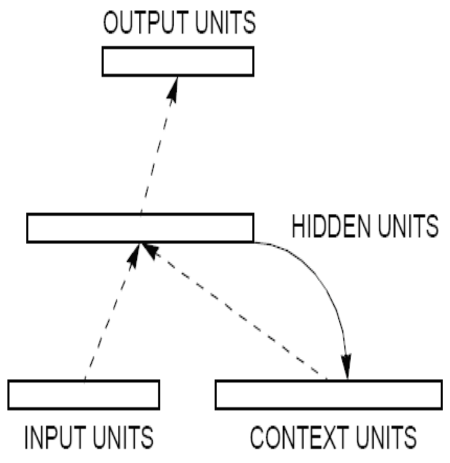

Fig. 4. A simple recurrent Neural Network [3].

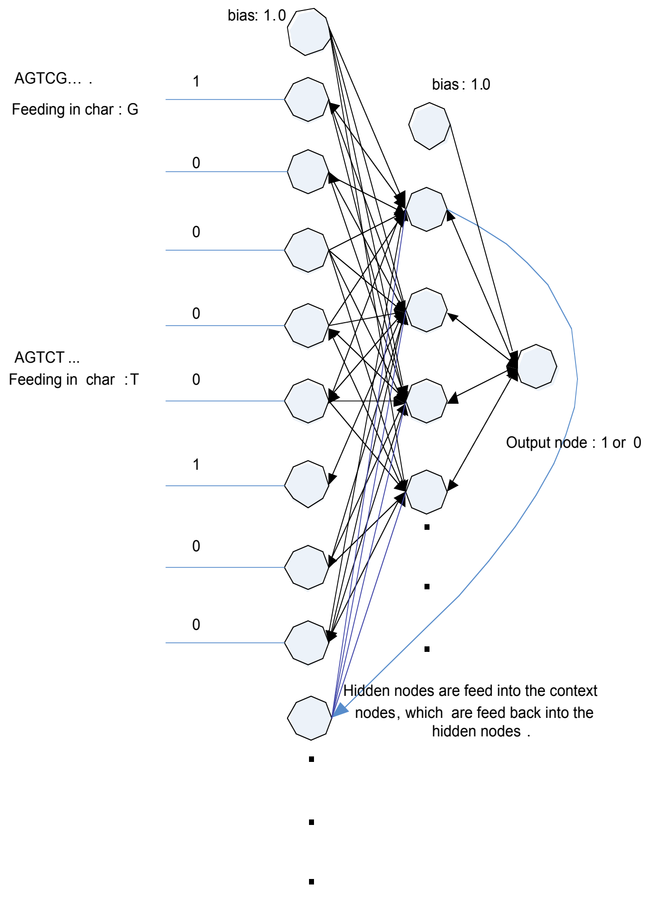

Fig. 5. The recurrent Neural Network that compares two sequences. Not all connections are drawn.

Training is performed after both sequences have been fed into the Neural Network. It is done by replacing the weights within the network. The weights are stored in a vector within each individual. There are 30 individuals in the population. Fitness is calculated on the same corpus for the whole population. It is increased if the individual scores above a threshold of +/-0.5. The sum of the squared error is also calculated. After all individuals are evaluated the population is ranked in the order of fitness. The worst individuals are replaced by randomly selected parents from the current population as illustrated in figure 6. Two points crossover is used with a variable mutation rate adjusted in the tuning phase. The mutation is calculated by a Gaussian distribution with a mean of zero and a variance of one. The mean squared error (MSE) for the best individual is calculated for each epoch as shown in figure 7. It is recorded in the training file for every epoch in the run.

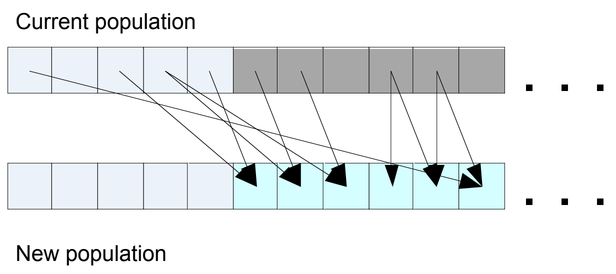

Fig. 6. Population replacement. Individuals in grey boxes are replaced by randomly selected parents in the current population. New ones in light blue boxes are produced.

	S1	S2	S3	. . .	**MSE**
I1	1.189	2.175	3.651	. . .	-
I2	1.056	2.955	3.250	. . .	-
I3	1.952	2.005	3.150	. . .	0.710
. . .	. . .	. . .	. . .	. . .	-

Fig. 7. An example showing the accumulated squared Error after each sequence (S) is evaluated for each individual in the population (I). MSE is calculated and recorded for the individual with the highest fitness score in the population.

The best individual in the population is validated against the performance of the best-saved individual in the same run by comparing the MSE between them. The better of the two individuals is saved into a data file. Validation prevents the network from over-fitting the data set, and ensures that the best individual is used for the tuning and "hold out" tests. The MSE for every epoch calculated from the best individual in the tuning and "hold out" test sets are recorded.

MSE does not determine whether the network generates the results correctly. Accounting for the percentage of correctness is a more accurate measurement of the network performance.

Both the number of individuals to keep during replacements and the mutation rate are adjusted only in the tuning phase from the first run. Initially, the number of individuals to keep is 10, and the mutation rate is 0.5. Tuning is not allowed in subsequent runs. The best-saved data uses a mutation rate of 0.5, and keeps the 10 best individuals.

III. Results and Analysis

Typically 31 runs should be produced for the results to be statistically significant. However due to time constraints, only 17 runs with 150 epochs each are presented in this paper. A different randomly generated seed is used for each of the 17 independent runs.

Data from all the training and validation sets are shuffled to prevent bias ordering of the data. Both the number of individuals to keep and the mutation rate are adjusted every 10th epoch after the first during the tuning phase. In order to save running time, the best-saved network is only tested every 10th epoch after the first.

The median performer of the 17 independent runs is determined by the histogram of the test performance from all runs as shown in figure 8. Frequency on the y-axis represents the number of runs that fall within a particular error range on the x-axis. The median performer is used to plot each of the three leaning curves in figures 9, 10, and 11. The initial randomized network is tested as epoch zero in the learning curves for the training, validation, and test sets.

The statistics and graphs for this experiment are generated by R, which is a language and environment for statistical computing and graphics [7].

The MSE is expected to drop after running more epochs for all three learning curves because the network should be able to make better generalizations from progressive trainings. Since only the best-saved individual from the run is used for testing, the test curve is flat on several intervals.

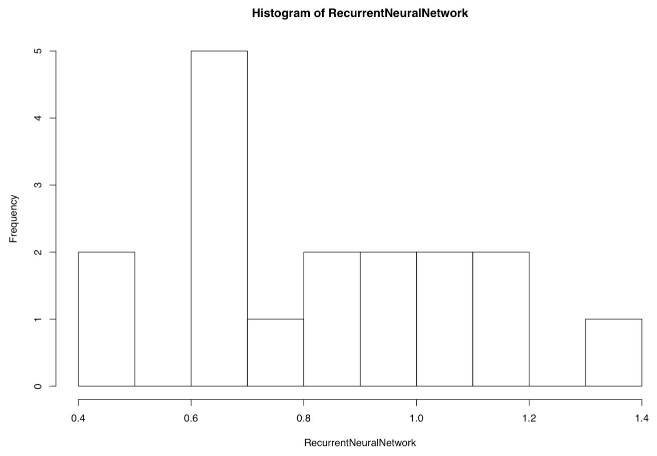

Fig. 8. Histogram of the performance of the recurrent Neural Network on comparing two sequences. The diagram shows the frequency for the mean squared errors from all 17 runs. The statistics from R are summarized as follows:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max
0.4793	0.6508	0.8197	0.8463	1.0160	1.3190

Standard deviation: 0.2532
Variance: 0.0641

The t test from R shows the 95% confidence interval for the true mean of the error distribution falls within [0.7161, 0.9765].

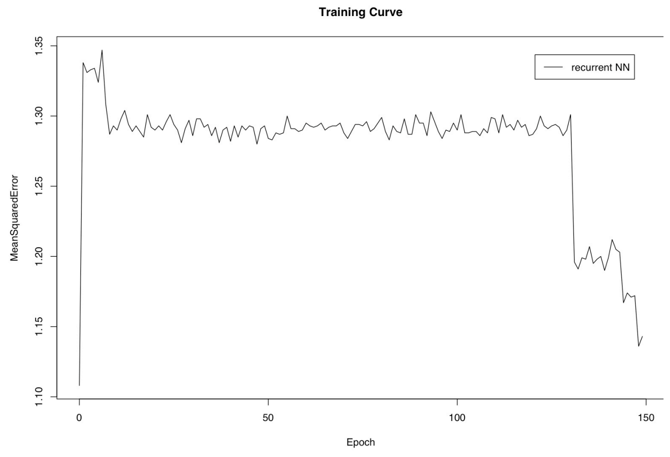

Fig. 9. Training curve for the recurrent Neural Network.

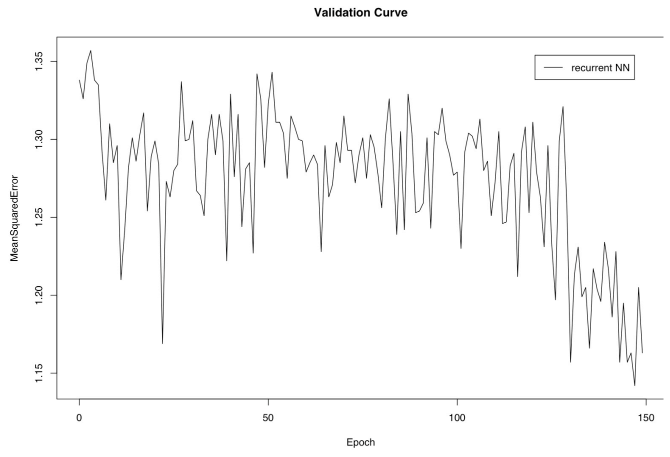

Fig. 10. Validation curve for the recurrent Neural Network.

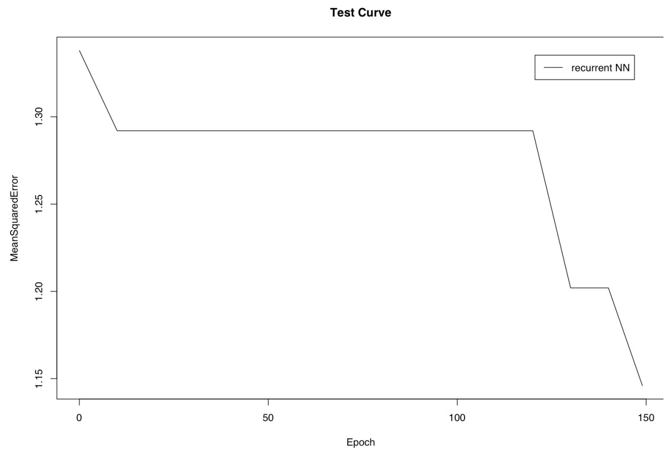

Fig. 11. Test curve for the recurrent Neural Network.

IV. Conclusion

Although many algorithms exist for genome sequence assembly, finding a more efficient algorithm is needed to assemble genome sequences that are thousands or millions of base pairs long. The process searches for overlapping regions among the set of sequences, and constructs the original sequence by combining these regions. As a result, it is critical to generate accurate sequence comparisons. This paper proposes using the Neuroevolution technique to compare two sequences of the same length. The preliminary results show that this technique is promising, but further modifications need to be made to account for the percentage of correctness. If the percentage of correctness increases during subsequent trainings, the network is expected to improve generalization on sequence similarity.

V. Future Work

The technique proposed in this paper is preliminary. It needs to account for the percentage of correctness. The work can be extended to handle comparisons between multiple sequences, and to account for more complex data sets including:

1) Sequences with deletions and insertions that should be classified as similar.
2) Sequences of different lengths.
3) Repetitive sequences.

Future modifications to the network include adjusting the number of hidden and context nodes, and refining the genetic parameters in the tuning phase. Furthermore, training can be done by Backpropagation instead of a GA.

Acknowledgment

I would like to thank Sara Nasser for her tutorial on bioinformatics and inputs on sequence alignment.

References

[1] Sara Nasser, Gregory L. Vert, Monica Nicolescu, Multiple Sequence Alignment using Fuzzy Logic, 2006.

[2] Mihai pop, Steven L. Salzberg, Martin Shumway, Genome Sequence Assembly: Algorithms and Issues, 2002.

[3] Jeffrey L. Elman, Finding Structure in Time. Cognitive Science, 14, pp. 179-211, 1990.

[4] Homo sapiens calcium channel from GenBank of NCBI, http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=nucleotide&val=134152371, Accessed April, 2007.

[5] Smith T. Waterman M, Identification of Common Molecular Subsequences, *Journal of Molecular Biology* 1981, 174, 195-197.

[6] NCBI / BLAST Home page, http://www.ncbi.nlm.nih.gov/BLAST/, Accessed May, 2007.

[7] R, http://www.r-project.org/, Accessed May, 2007.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Steins, Introduction to Algorithms, Second Edition, 2001, pp. 321-324.

[9] Zheng Zhang, Scott Schwartz, Lukas Wagner, Webb Miller, A Greedy Algorithm for Aligning DNA sequences", Journal of Computational Biology, vol7, pp. 203-214, 2000.

[10] Kaplan K. M, and Kaplan J. J., Multiple DNA sequence approximate matching, Computational Intelligence in Bioinformatics and Computational Biology, vol. 7, pp. 79-86, 2004.

[11] Niko Välimäki, Wolfgang Gerlach, Kashyap Dixit and Veli Mäkinen, Compressed suffix tree – a basis for genome-scale sequence analysis, Bioinformatics vol. 23, number 5, pp. 629-630, 2007.