DAVI: Deep learning-based tool for alignment and single nucleotide variant identification

IOP Publishing Mach. Learn.: Sci. Technol. 1 (2020) 025013 https://doi.org/10.1088/2632-2153/ab7e19

MACHINE LEARNING
Science and Technology

DAVI: Deep learning-based tool for alignment and single nucleotide variant identification

G Gupta[1,2] and S Saini[1]

[1] Indian Institute of Technology Hauz Khas, New Delhi, Delhi 110016, India
[2] Author to whom any correspondence should be addressed.

E-mail: ggupta.iitd@gmail.com **and** saini.shubhi@gmail.com

Keywords: genome, deep learning, single nucleotide variant caller, genome-assembly, pipeline, convolutional neural network, recurrent neural network

OPEN ACCESS

RECEIVED
22 October 2019

REVISED
14 January 2020

ACCEPTED FOR PUBLICATION
9 March 2020

PUBLISHED
27 May 2020

Abstract

Next-generation sequencing (NGS) technologies have provided affordable but errorful ways to generate raw genetic data. To extract variant information from billions of NGS reads is still a daunting task which involves various hand-crafted and parameterized statistical tools. Here we propose a deep neural networks (DNN) based alignment and single nucleotide variant (SNV) identifier tool known as DAVI: deep alignment and variant identification. DAVI consists of models for both global and local alignment and for variant calling. We have evaluated the performance of DAVI against existing state-of-the-art tool sets and found that its accuracy and performance is comparable to existing tools used for bench-marking. We further demonstrate that while existing tools are based on data generated from a specific sequencing technology, the models proposed in DAVI are generic and can be used across different NGS technologies as well as across different species. The use of DAVI will therefore help non-human sequencing projects to benefit from the wealth of human ground truth data. Moreover, this approach is a migration from expert-driven statistical models to generic, automated, self-learning models.

1. Introduction

Next-generation sequencing (NGS) [1] has opened up a new paradigm - it has provided affordable access to whole genome or genome sequences to many researchers, which leads to personalized medicine and to personal genome projects [2]. Many Mendelian disease studies have employed NGS to identify causal genes based on patient-specific variants [3]. Since a disease-associated genetic variant rarely occurs among the general healthy population, its interpretation in a patient is relatively simple. This interpretation simplicity by virtue of rarity of the variants, however, entails a risk of false discoveries due to the errors in sequencing and false detection by variant calling methods. For the success of clinical genomics and personalized medicine, fast and accurate identification of variants is crucial. Thus, NGS technology has shifted its focus from generating genome data to swift and accurate information extraction. In NGS methods, a whole genome or targeted regions of the genome are randomly digested into small fragments (or short reads) that get sequenced and are then either aligned to a reference genome or assembled [4] to form complete DNA (de-novo assembly). The Illumina HiSeq2000 sequencer generates reads with length 90–110bp with a sample frequency of 30x and read error at 1%, whereas PacBio RSII generates long reads with a median length of 20kbp but with higher indel error rate of 15–20% of read length. Data produced by sequencers are then analysed by finding overlapping regions called contigs and then lining them up. The alignment of reads is a computational challenge, as reads do not contain position information; that means we have to use sequence information only to find the corresponding region in the reference sequence. The reference sequence can be quite long (3 billion bases for human), making it a daunting task to find a matching region. Moreover, short reads may be getting aligned to several equally likely places. This is especially true for repetitive regions. Then to segregate reads errors from potential variants statistical models are used. The variants thus identified are known as true variants and the process of detecting variants in reads is known as variant calling. Since NGS represents a throughput technology, it is highly sensitive to technological errors and produces data which is erroneous, random, multiple and of high coverage. It is

therefore necessary to clean and pre-process [5] [6] [7] [8] data before variant calling. Variant detection in a genome sequence is therefore a multi-step process comprising of: (1) mapping the reads to the indexed reference (alignment); (2) sorting the reads based on their location; (3) identifying and annotating the probable variant candidates; and (4) determining true variants and their genotypes from variant candidates with high confidence (variant calling). This makes the process of variant and indels (insertion and deletion of bases) identification highly dependent on reliable bio-informatics tools. There are many off-the shelf tools and pipelines available for variant identification [9]. These tools are based on string manipulation and statistical modeling, and require parameter tweaking by domain experts. Moreover, different tool sets are required for processing data from different NGS technologies and for different species.

In this paper, we propose a novel deep neural network (DNN) based tool "DAVI" (deep alignment and variant identification), which can be used across NGS technologies and species for raw reads alignment and single nucleotide variant (SNV) detection. We benchmark the computational performance of DAVI against state of the art GATK pipeline and demonstrate that it not only performs variant identification faster, but is also more specific than existing tool sets.

2. Current methodology

As described above, all existing pipelines can be broadly divided into two major processes: the first is aligning reads to reference **(alignment)**, and the next is identification of variants from aligned reads **(variant calling)**.

Alignment is the process by which the NGS reads are aligned to their corresponding (most likely) locations in the reference genome. Alignment can be achieved by comparing reads with k-mers of the reference genome. But due to imperfection in raw NGS reads (substitution and indels) an attempt to exactly match reads to reference k-mers will lead to rejection while comparing. Thus to perform alignment, raw reads are divided into sub-reads and matching regions (contigs) with reference are identified. Matched sub-reads are expanded at contigs to find the best suitable match of a read. There exist many tools which perform this local alignment, such as BWA, Novalign, Bowtie, BLAST, etc. These tools use dynamic programming algorithms to locate the region for best alignment. Algorithmically these tools can be divided into two main categories : (1) hash table-based algorithms, indexing either the reads or the reference genome; and (2) Burrows-Wheeler transform based algorithms, using suffix trees and suffix arrays of the strings [10, 4]. Suffix array-based aligners are memory-efficient and work faster than hash-based aligners, but they are less accurate. In contrast, hash table-based algorithms tend to be slower, but more sensitive. In recent times, substantial work has been done to incorporate machine learning techniques in the field of bioinformatics. There are many methods which use models to encode high-dimensional genomic data into vector data and then use machine learning algorithms for alignment and motif generation. BioVec and ProtVec [11] use a skipgram-based model to represent protein sequences and their classification. Aoki and Sakakibara [12] have used Word2Vec encoding to do alignment and generate motif of non-coding regions of RNA. Nucl2Vec [13] also uses encoding similar to word2Vec and uses KNN algorithm to do alignment of sequencer data.

Variant calling is the process by which variants are identified from the aligned sequence data with respect to reference sequence of the particular organism. GATK, VarScan2, Atlas-SNP2 and SNVer are some of the tools that perform variant calling on aligned reads. SNP callers may also be divided into two different approaches: (1) heuristic methods based on thresholds for coverage, base quality and variant allele frequency; and (2) probabilistic methods based on genotype likelihood calculations and Bayes' theorem. Due to their computational demands, heuristic-based methods are less commonly used than probabilistic methods [4]. Some studies have been conducted which use machine learning techniques for indel identification using random forests [14]. But still GATK is the de facto industry standard for identification of SNVs in Illumina datasets.

2.1. GATK best practices pipeline

GATK best practices pipeline [15] is the most widely used method for SNV calling, as it gives the best performance across existing variant calling pipelines against the gold standard set of reference from GIAB [9]. This pipeline consists of BWA, Picard Tools and GATK packages. For the purpose of benchmarking the performance of our proposed SNV identification methodology, we have experimented with Illumina2000 data and compared our results with the GATK best practices pipeline.

GATK uses machine learning-based algorithms in its HaplotypeCaller tool for variant detection, with the PairHMM method for pairwaise aignment of each read against its haplotype. PairHMM is a pairwise alignment method that uses a Hidden Markov Model(HMM) and produces a likelihood score of observing the read, given the haplotype [16]. For each potentially variant site, the program applies Bayes' rule, using

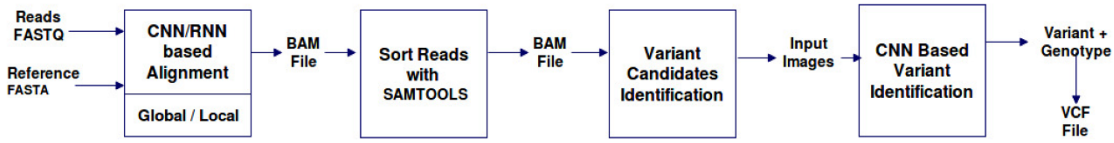


Figure 1. DAVI pipeline.

the likelihoods of alleles given the read data to calculate the posterior likelihoods of each genotype per sample given the read data observed for that sample.

2.2. Deep learning-based approaches in bio-informatics

Despite more than a decade of effort and thousands of dedicated researchers, the hand-crafted and parameterized statistical models used for variant calling still produce thousands of errors and missed variants in each genome [17]. DNN-based learning algorithms provide attractive solutions for many bio-informatics [18] problems because of their ability to scale for a large dataset, and their effectiveness in identification of intrusive complex features from underlying data. There has been some success in area of de novo peptide sequencing [19], mapping protein sequence to fold(deepSF) [20] and to predict protein binding sites in DNA and RNA(deepBind) [21]. Convolutional neural network (CNN)-based models are used exhaustively in identification of motif in DNA sequence [22], but not much has been done for the prediction of SNVs in raw DNA reads.

Only recently, Google has developed the DeepVariant [17] tool, which uses deep learning to predict variants in aligned and cleaned DNA reads. The DeepVariant method uses CNN as a universal approximator for the identification of variants in NGS reads. It does so by finding candidate SNPs and indels in reads aligned to the reference genome with high sensitivity but low specificity. The DeepVariant model uses Inception-v2 architecture to emit probabilities for each of the three diploid genotypes at a locus using a pileup image of the reference and read data around each candidate variant. There have been some attempts to use Recurrent neural network (RNN) for global alignment, but their results are not yet established [23].

In DAVI we have used both CNN and RNN algorithms for alignment. Using DNN we determine a DNA read comparator function (global alignment) or sub-read comparator function (local alignment), which determines the best alignment based on a scoring function. For variant identification DAVI uses a CNN like DeepVariant but instead of using pileup images with inception-v2 architecture, we use a position-specific frequency matrix (PSFM) to identify possible variant sites and a set of images (reference, substitution and indel) to predict a variant and its genotype at a given location. The detailed model designs and experiments are illustrated in the section below. We have used Illumina data of *E. coli* (*CP012 868.1 Escherichia coli str. K-12 substr. MG1655, complete genome* [24]) and human data (*GRCh37/hg19.chr20* [25]) for validation of our experiments. We have also used pacBio human genome data (*hg38.chr21 & hg38.chr22* [26, 27]) for generalizing our model across NGS technologies.

3. Deep aligner and variant identification

3.1. Alignment

As discussed above, to perform alignment we have to derive a comparator function, which considers complete raw NGS reads (global alignment) or part of reads (local alignment), and reference k-mers from a location as input, and performs comparison to output rank of similarity, since perfect matches to the reference sequence are elusive. Our comparator function should be able to handle some percentage of variation between two input streams while measuring the degree of similarity. Higher similarity value implies better alignment at that location.

As we know, DNN works very well in extracting features while being resilient over noise(variations). Moreover, since DNN-based algorithm scales with data, our alignment use-case is a good fit for these algorithms. The two most widely used DNNs are CNN and RNN. While CNN is generally used to extract features from local patches of data, RNN is used for sequential data where the current outcome is dependent upon previously learnt patterns. We have explored both these techniques to solve our problem of alignment. The dataset used for training and testing of our DNN model is discussed in following subsection.

3.2. Dataset

To perform alignment we have used the genome sequence of *Ecoli K-12* [24]. For this reference genome we have used two sets of input reads: (1) Actual NGS reads, which were generated from the ILLUMINA

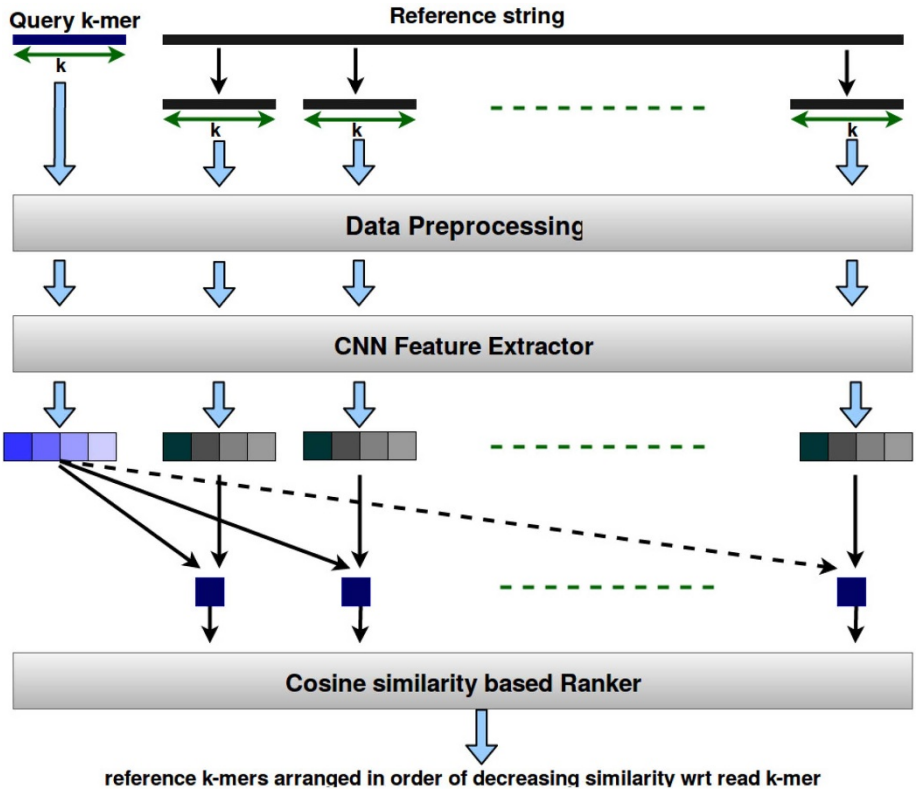


Figure 2. Deep Conflation Based Model.

sequencer and have a length of 30–300bps with error rate of 1–2% and coverage of 30x. We called this dataset real dataset. We have used this dataset primarily for testing our models. (2) To learn a robust comparison function we have trained our models with what we call a simulated dataset. Reads for this dataset are generated from the reference genome with a max error rate of 40%. The process of generation of simulated reads and preprocessing of strings for alignment is explained in appendix B.

3.3. DNN for alignment

3.3.1. Deep conflation-based model

In this model, we have used CNN to predict the most probable location of a given read by breaking the reads and reference into fragments of size k, called k-mers. Let r_i be a k-mer obtained from reference location i. We calculate the similarity between k-mer of a given read w.r.t. several different reference k-mers from different locations $\langle r_1, r_2, .., r_i, ..., r_n \rangle$. The highest similarity reference k-mer can then be extended in a fashion similar to the BLAST algorithm. We formulate the problem of k-mer string matching as analogous to that of identifying match between a word and its misspelled variants. This formulation is similar to the conflation problem of business data. 'Character-level Deep Conflation For Business Data Analytic' [28] discusses the model developed to solve the conflation problem. We have extended this model for our application. The details of the model used for alignment are discussed in appendix B. Figure 2 gives an overview of the model.

3.3.2. Evaluation on simulated dataset

To perform alignment using this model, a simulated dataset was generated from the genome sequence of *E. coli* (*CP012 868.1 E. coli str. K-12 substr. MG1655, complete genome* [24]). We experimented by generating a set of 10 000 random 100-mers containing characters A,C,G and T. For each 100-mer, we also generated a 100-mer with 40% insertion, deletion and substitution errors starting at a random location. After training the model for three epochs on the entire dataset, with mini-batches of size 100, the model was able to achieve a testing accuracy of 99.4%.

The entire dataset was divided into training and testing data, using a split ratio of 90:10 for training:testing. The training data was further split in a ratio of 90:10 for training and validation, with data selected through random permutation on the dataset for each epoch. With this splitting, each epoch contains 81 minibatches. Figure 3 shows the change in loss and accuracy of prediction over the number of batches

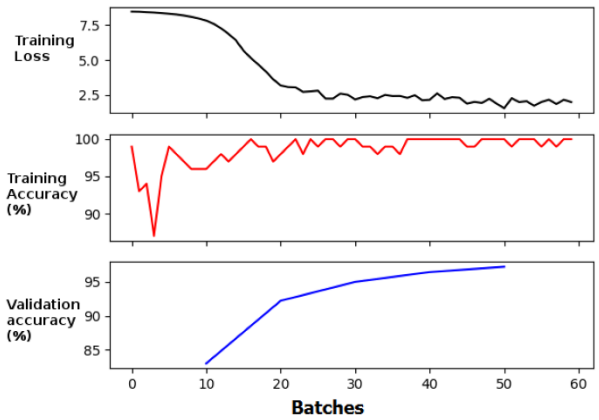


Figure 3. Loss vs number of batches.

Table 1. Performance Metrics of Deep Conflation Based Alignment Model.

Metrics	Values
Accuracy	0.995 323 901 984
Precision	0.949 205 079 492
Recall	0.999 578 814 362
F1 Score	0.973 7 901 934

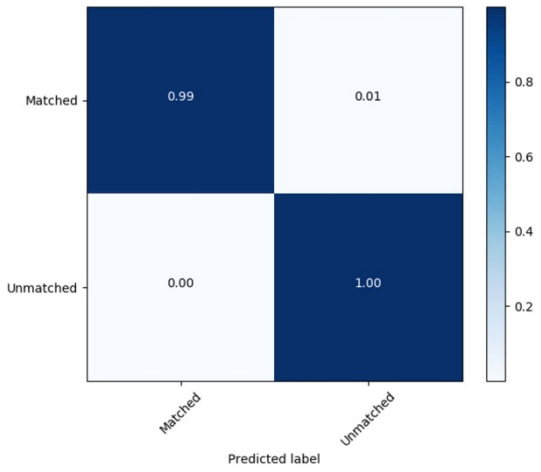


Figure 4. Confusion Matrix for Deep Conflation Alignment Model.

during training and validation. Accuracy for validation and testing is measured as the number of mutated strings for which the correct reference was ranked highest in the given set.

3.3.3. Evaluation on real data

To test the accuracy of a network trained with the above simulated dataset with real data, the following procedure was used. We have taken an *E. coli* reference the same as that used in simulated data and NGS reads from *SRR2724 094_1.bam* (fastq aligned by BWA-MEM). The process is defined in Algorithm 3 (appendix A). The model was tested on 100 batches of the above dataset where the input sequence length is of 100 nucleotide bases. Statistics of the test are shown in table 1 and figure 4.

3.3.4. Observations

The Deep Conflation model has been found to be useful for predicting the best match reference k-mer to a given read. In this model the length of read (k-mer) does not affect the accuracy of prediction. Thus this model can be used for global alignment, since a given read of length 'r' can be aligned by searching for the best reference k-mer of the same length by straightforward comparison. Since feature extraction for

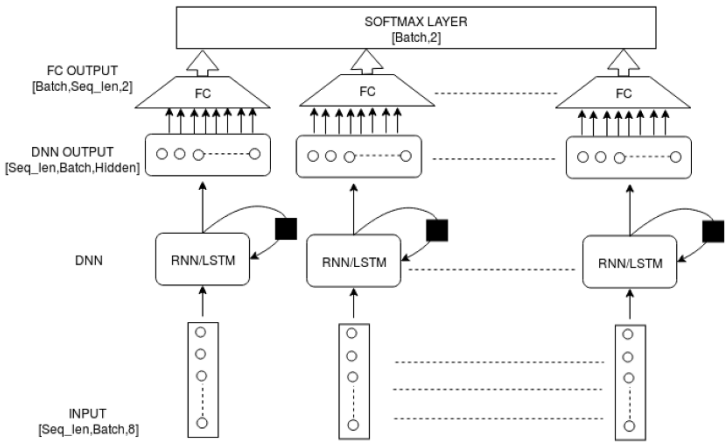


Figure 5. RNN Based Alignment Model Architecture.

Table 2. Vanilla RNN and LSTM Hyper Parameters.

Hyper Parameters	RNN	LSTM
Batch Size	10	10
Input Length	10–90	10–90
Hidden State Size	4	20
Bias	True	True
Dropout	0.0	0.0
Learning Rate	0.001	0.003
Optimizer	RMSProp	Adam
Loss Function	Cross Entropy	Cross Entropy

reference k-mers is required only once, we create a database of reference features first time. Subsequently, to test the alignment of reads against this reference, the features are queried from the database.

3.4. RNN based alignment

RNNs are memory-based neural networks. They are especially useful with sequential data because each neuron/unit can use its internal memory to maintain information about the previous input. The two most widely used RNNs are Vanilla RNN and LSTM (Long Short Term Memory). RNN is estimated to be a faster network as compared to LSTM, but it falls short in memorizing and representing long-term patterns associated with input data stream. Thus, to determine which RNN will be feasible for comparing genome strings, we modeled both Vanilla RNN cells and LSTM cells. We further optimized hyper-parameters of these models using *CoDeepNEAT* techniques.

3.4.1. Architecture

Alignment of query reads with respect to a given reference sequence works by comparing query string with a series of reference strings, and determining the degree of confidence of their alignment. Since a read may be aligned to more than one location in the reference, 'degree of confidence' gives us a measure to choose the best site for alignment. To perform alignment using neural network, our RNN takes reference string and query string as input and then classifies them into one of the two classes: namely, matched class and unmatched class. Then from all matched cases based on best matching score, a reference string is picked for alignment. For classification, we have modeled our DNN as shown in figure 5. The values of hyper-parameters used in Vanilla RNN and LSTM model are listed in table 2.

3.4.2. Experiment

Models of both RNNs are trained on Simulated Dataset. Input for the models is generated by concatenating a raw read sequence of fixed length with a reference sequence of the same length against which the comparison is to be made. This concatenated sequence is then converted to Hot Vector Encoding before being fed to the models. For Vanilla RNN 50 epochs with 380 batches of a training dataset are used, while for LSTM 100 epochs are used for training the same dataset. To observe over-fitting, a validation test was performed on the validation dataset at regular intervals.

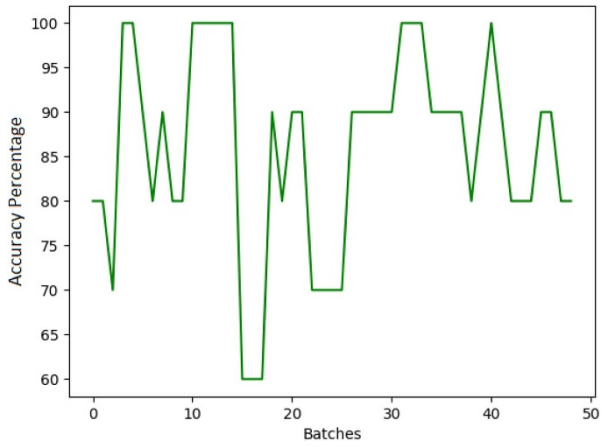


Figure 6. Vanilla RNN Model Accuracy On Test Dataset.

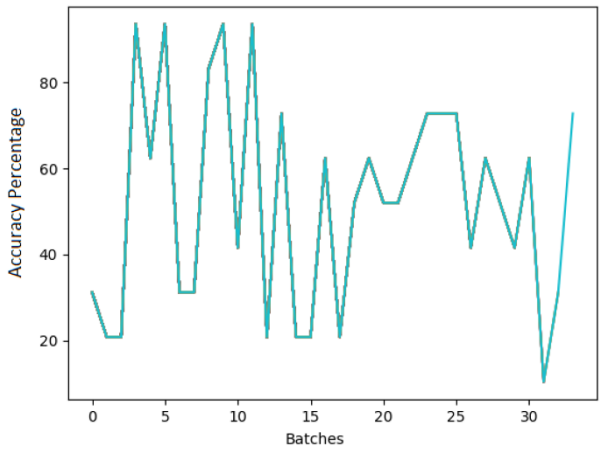


Figure 7. LSTM Model Accuracy on Test Dataset.

3.4.3. Evaluation and observations

A trained RNN model was evaluated on a test dataset of simulated data. A mean accuracy of 87.55% was observed for Vanilla RNN while for LSTM the mean accuracy was 79.6%. The accuracy graph of Vanilla RNN and LSTM on the test dataset are shown in figure 6 and figure 7, respectively. The accuracy of a batch is found to be in negative correlation to the percentage of errors in sequences. This means batches where large number of sequences have high error show low accuracy. This was observed for both Vanilla RNN and LSTM-based networks. Detailed Design for both Vanilla RNN and LSTM with training accuracy is illustrated in appendix C.

3.4.4. Comparison of RNN models

We have performed experiments with different sequence lengths ranging from 10 to 90. The number of hidden units in the models is kept at 1/3 of the sequence length. We have observed that as the size of the sequence grows beyond a certain threshold, both RNN models show no learning and accuracy drops significantly. For Vanilla RNN, the model threshold is reached for a sequence length above 50, while for LSTM the model threshold is reached at a length of 80. The test accuracy of the Vanilla RNN model is significantly higher than the LSTM model, and also the Vanilla RNN model trains faster as compared to the LSTM model. The test accuracy of both models vs sequence length is shown in figure 8.

As observed, the Vanilla RNN model has better accuracy as compared to LSTM for query length up to 40bps; we have used Vanilla RNN for alignment. Since the query length of NGS data is more than 40bps, the Vanilla RNN-based model can be used for local alignment only. It is interesting to note that the LSTM-based comparator accuracy degrades for reads of longer length. It may be because as length increases the amount of error (max 40% of total read length) in simulated read increases, and LSTM is not robust to noise; this degrades its classification accuracy.

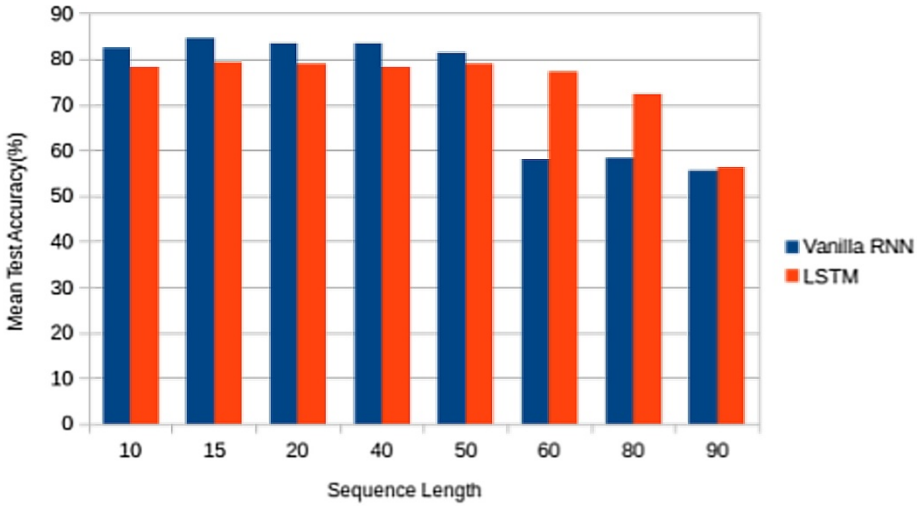


Figure 8. Test Accuracy Comparison of Vanilla RNN and LSTM Model with respect to Input Sequence Length.

3.5. Optimization of RNN models

The above two experiments are clear indicators that Vanilla RNN-based neural network does perform well for genome comparison, but designing an optimal RNN network is a difficult task as each network has a large number of hyper-parameters to optimize. To automate the process of finding best values for each hyper-parameter, we have used the neuro evolution technique known as CoDeepNeat [29]. CoDeepNeat uses existing neuro evolution technique of NEAT [30], which has been successful in evolving topologies and weights of relatively small recurrent networks. The fitness of evolved DNN is determined by how well the system is getting trained by gradient descent, to perform its task. CoDeepNeat uses a genetic algorithm for optimization, which is combined with gradient descent to evolve large and complex DNN.

3.5.1. Extending CoDeepNeat for RNN

In our implementation of CoDeepNeat, population of chromosomes is created by randomly initialized hyper-parameters of the network and slowly evolving them though mutation. Instead of representing each chromosome as a layer in DNN, in our design each chromosome itself is represented as a DNN.
The advantage of this modification is that it nullifies the requirement of crossover in genetic algorithm as crossed over offspring can be generated by the application of mutation only. Although this approach limits the size of DNN and the number of different layers that can be stacked, yet this is not a constraint since our network is small. To evolve the networks, we have used two different mutation operators, namely Random Mutation Operator and Gaussian Mutation Operator [31]. To evolve population offspring with selection, we have used DNN learning as a fitness function, in particular, the accuracy of the validation test of DNN after a fixed number of epochs. Chromosomal population is evolved using a genetic algorithm. The algorithm used for evolving chromosome population selects three elites from the parent generation and these elites are evolved in the next generation without mutation, while all other offspring are mutated versions of previous generation chromosomes. The detailed algorithm for evolution is described in Algorithm 4 (appendix A).

3.5.2. Evolution and training

Evolution was carried out for 10 generations (epochs) and the best DNN was selected as the one with the highest accuracy in any generation of evolution. To mutate chromosomes, mutation operators were applied to each hyper-parameter with a probability of 0.2. Since training a DNN is computationally expensive and population size was 30, each network was trained for only two epochs on the training set. Validation set was used for calculating the fitness of the DNN. Due to the small number of epochs, validation accuracy was calculated once, at the end of training. It is argued that due to the small number of epochs, the network which is evolved is the one which trains fastest rather than the one with highest accuracy. Thus, through this technique we have automated the design of DNN topologies, which gives the model with shortest training time with relatively higher accuracy.

3.5.3. Experimental setup and evaluation

The system is built using *Pytorch* libraries with *Tensorflow* backend for CUDA platform on single GPU. The time taken for the model to evolve is 49.32 hrs. After the network is evolved for 10 generations, for each DNN

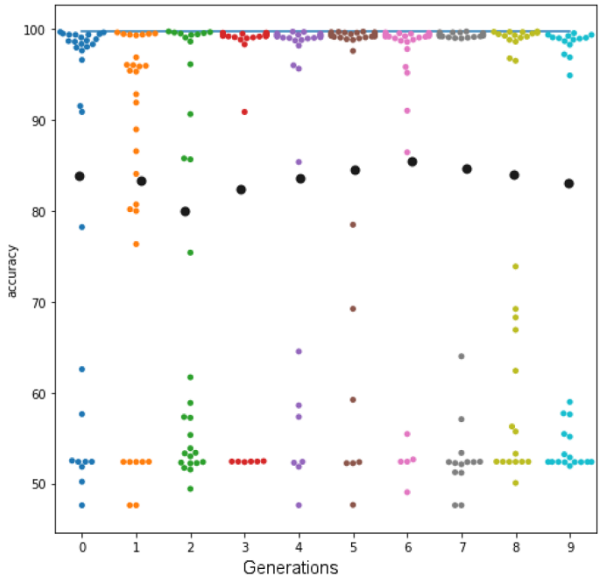


Figure 9. Accuracy of Population after each generation.

Table 3. Hyper Parameter Values of Suggested RNN.

Hyper Parameter	Values
Number of Layers	1
Learning Rate	0.093 8 099
Optimizer	RMSProp
Bias	False
Hidden Size	7
Dropout Rate	0.265 3 750 429 101 209
Activation Function	sigmoid
Bi-directional	False

(chromosome) in population, mean validation test accuracy is calculated. Figure 9 shows the performance of the population across generations. It is clear from this figure that the system evolves until the 6th generation by increasing mean accuracy and minimizing the standard deviation of the population. But after the 6th generation, mean accuracy decreases due to random mutation and standard deviation of population increases. During evolution most of the DNNs (population) trained quickly and had accuracy between 90%–100% as shown by the density of clusters in figure 9. For the optimal solution, we chose DNN from the 6th generation which provides maximum accuracy. The DNN with the highest accuracy and smallest standard deviation has been picked up as the best DNN for nucleotide base comparison. The values of hyper-parameters of the best DNN suggested by the model are listed in table 3.

3.5.4. Evaluation of optimized RNN with simulated data

To train and test the optimized RNN model, a simulated dataset is used. The training loss graph (figure 11) of the optimal DNN converges faster as compared to the Vanilla RNN. Vanilla RNN learning starts at around 3500 batches, while in optimal DNN the learning starts just after 2500. Hence, the number of batches required for training the optimal DNN is much less than the Vanilla RNN model. Mean test accuracy of the optimized RNN for 100 samples of simulated dataset is 96.123 % (figure 10).

When tested on a real dataset with input sequence of length 15 nucleotide bases, mean accuracy for 100 samples increases to 98.88. The accuracy plot and report are shown in figure 12 and table 4.

3.5.5. Limitation and assumptions

Limitations of the RNN based model are as follows:

- We have assumed the reads to be of fixed length and maximum length of a read is less than 40 bps.
- This model falls short in doing global alignment, as length of read generated by Illumina NGS are greater than 40 bps.
- We have not considered quality of reads while performing comparison.
- We have assumed maximum errors in high-quality reads are 40% of total length.

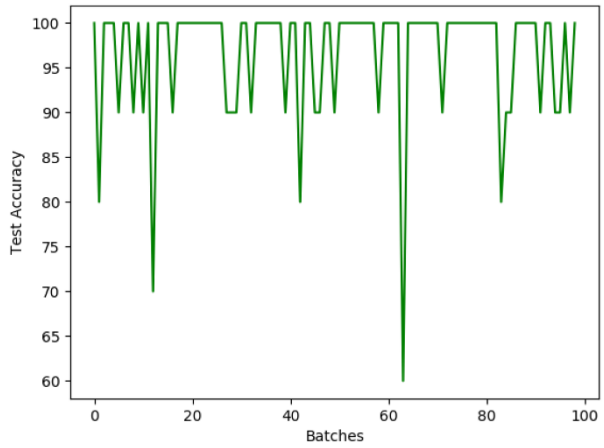


Figure 10. Test Accuracy of Optimized RNN.

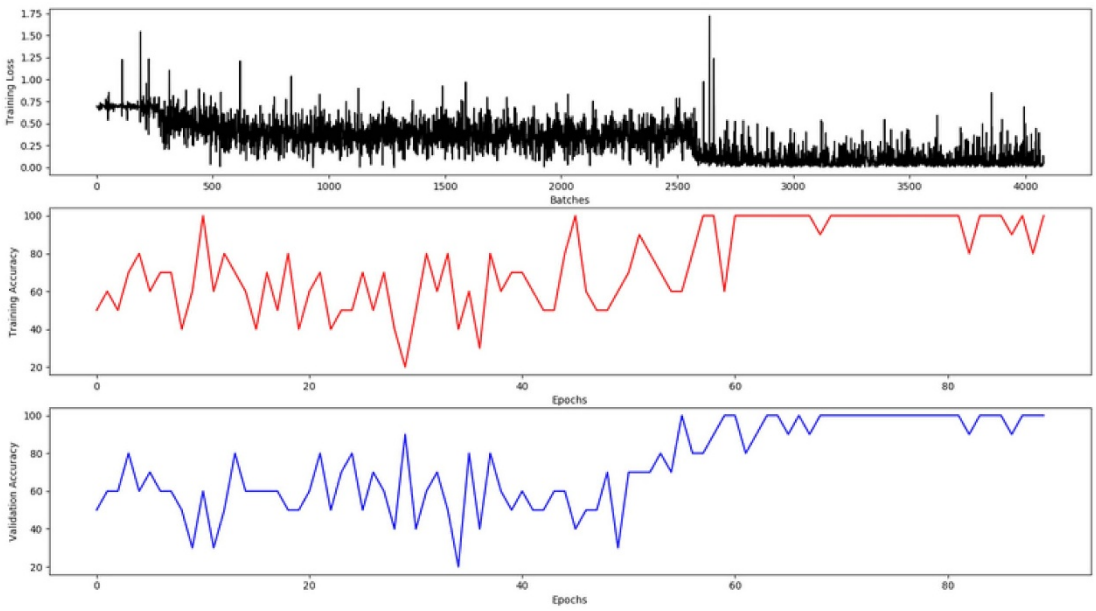


Figure 11. Performance of Optimized RNN.

4. Using DNN models for alignment

We have described CNN and RNN-based models which can be used to generate comparator function for alignment. These models need to be discussed from the viewpoint of their usage in the alignment framework. The RNN-based alignment model has been found to work well on k-mers with a length less than 40bps. Since the reads produced by Illumina are greater than 40bp in length, this model is more suited for local alignment of reads. To align a read w.r.t. the reference genome, RNN has to be used in consultation with the local alignment algorithm as described in Algorithm 5 (appendix A).

To increase the speed of alignment, we have used heuristics similar to BLAST to reduce our search space, by reducing the number of comparisons. Instead of comparing with each reference k-mer, the algorithm uses 5-mers as seed, and based on the seed all possible locations are searched for alignment. To reduce search space for query sequence, a database of all possible 5-mers with their locations in the reference genome are stored in dictionary format. Once a database is created, all possible 5-mers in query sequence are read. These query 5-mers are then searched in the database and their corresponding location of occurrences are saved as probable locations for alignment. The detailed work flow of the algorithm is as described in Algorithm 1.

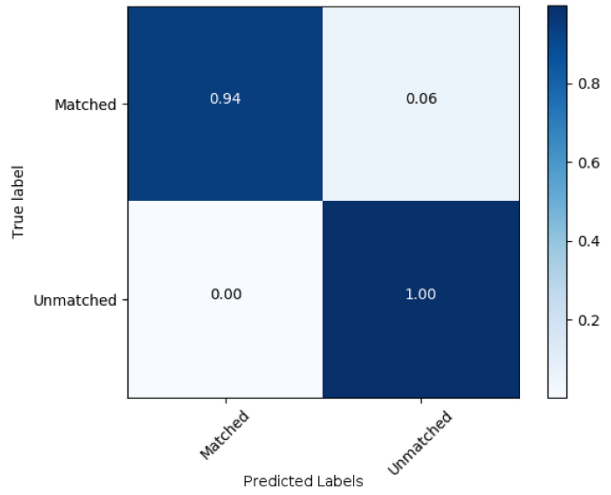


Figure 12. Confusion Matrix Of Optimized RNN On Real Dataset

Table 4. Performance Metrics of Optimized RNN Alignment Model.

Metrics	Values
Accuracy	0.988 3 333 333 333 333
Precision	0.990 2 534 113 060 428
Recall	0.996 078 431 372 549
F1 Score	0.993 157 345

Algorithm 1 Alignment with Heuristic.

procedure createDatabase(referenceSequence,kmerSize)

▷ Create reference kmer database

kmerDatabase ← *dict*()

▷ create dictionary key:kmer ,value: location array

refLength ← *lengthOfSeq*(*referenceSequence*)
while $i \neq (refLength - kmerSize)$ **do**
kmerWord = *referenceSequence*[$i : i + kmerSize$]
$i \leftarrow i + 1$
if kmerWord in kmerDatabase.keys() **then**
kmerDatabase[*kmerWord*].*append*(*i*)
else
kmerDatabase[*kmerWord*] ← [*i*]
SavetoFile(kmerDatabase)

▷ Save kmer database to file

procedure Alignment(queryRead,referenceSequence,type,kmerSize)

▷ Aliment procedure with heuristic

refKmerDatabase ← *readfromFile*(*kmerDatabase*)

▷ Read reference kmer dict from file

queryLength ← *lengthOfSeq*(*queryRead*)
refSequenceSet ← []
while $i \neq (queryLength - kmerSize)$ **do**
queryKmer ← (*queryRead*[$i : i + kmerSize$]
locSet ← *refkmerDatabase*[*queryKmer*]
for each location in locSet **do**
start ← *location* − *i*
end ← *start* + *queryRead*
refSequenceSet.append(referenceSequence[start:end])
if type is Local **then**
alignedlocation ← *localAlignmnet*(*queryRead*, *refSequenceSet*, *kmerLength*, *batchSize*)
else
alignedlocation ← *GlobalAlignment*(*queryRead*, *refSequenceSet*)
return *alignedlocation*

Table 5. Alignment models timing comparison.

Model	Training Time	Prediction Accuracy	Alignment Type
Deep Conflation Model	4 hours	99.5	Global
RNN Model	14.24 min	98.8	Local

Table 6. Time taken (in sec) by DNN based alignment models to predict location of Read.

Read Name	Length of Read	RNN	Deep Conflation	BLAST	BWA-MEM
SRR2724094.1.1 1	300	4.34	1.8	10.85	0.64
SRR2724094.3.1 3	37	0.624843	0.15	0.169	0.045
SRR2724094.7.1 7	86	0.377	0.284	0.876	0.047
SRR2724094.10.1 10	168	1.548	0.603	0.72	0.055

Table 7. Reads location prediction by different alignment models.

Read	BLAST		RNN		Deep Conflation	
	Location	Score	Location	Score	Location	Score
SRR2724094.1.1 1	—	—	4258852	0.388	4258852	0.389
SRR2724094.3.1 3	3073798	0.945	3073797	0.95	3073797	0.2
SRR2724094.7.1 7	306992	0.97	306991	0.98	306991	0.98
SRR2724094.10.1 10	1564588	0.87	1564587	0.95	1564587	0.95

4.1. Comparison of models

We have experimented with *E. coli* data [32] on deep conflation and RNN models as described in algorithms 1, 3 and 5. The *E. coli* reference used was the same as that across all alignment models, and reads were obtained from sample *SRR272 401_1*. The models were compared for time taken by them to train and predict accuracy, and results are summarized in table 5.

The models proposed for alignment were also compared with existing state-of art alignment tools, mainly with BLAST and BWA-MEM. For BLAST [33] we have taken open source python implementation, while for BWA-MEM the standard tool was taken for comparison. The result of performance comparison is shown in table 5. It is observed that BWA-MEM is highly optimized for alignment and it is fastest irrespective of query length. BLAST is also optimized by using several heuristics to limit the search space. Our proposed model Recurrent Aligner works well for query of small length as the model is designed for an optimum length of 32bps, while the deep conflation model works well for query length based on training configuration. When trained for 100bps sequences, the deep conflation model works well for sequences 100bp length, but is unable to align queries shorter than 50 bps. On training the model with 30bps sequences, queries of 30bps are aligned correctly. Therefore the deep conflation model is not able to align queries with more than 50% of padding. For high-quality alignment, the threshold value of the score can be set in the proposed model.

4.2. Deep learning-based variant calling

The SNV and indel identification in a genome is a complex problem, which can be broadly broken down into two subproblems: identification of variant and classification of their type. Probable variant sites can be classified into four categories : heterozygous, homozygous, non-variant and non-SNP. Since we have already discussed how a CNN-based deep learning model can be used for extracting features and for classification of genomic data, we have used CNN for identification and classification of variants. The proposed variant caller pipeline is as described in figure 13. Data is preprocessed where from aligned and sorted reads probable variant candidates are located using a position frequency matrix. Each variant candidate is converted to input image matrix, which is fed to trained CNN for classification of probable variant candidates to variant/non-variant and their genotype. The details of data processing, input image creation and label data generations are described in appendix D.

For training the CNN model, we have used two sets of datasets, one of human genome and another of *E. coli*. For human, we have used the Genome in a Bottle reference dataset [34]. Out of this dataset we have used *NA12878* [35] data, in which input data is in BAM format while variant data is in VCF format. To consider a variant which has high confidence, interval data was used in BED format.

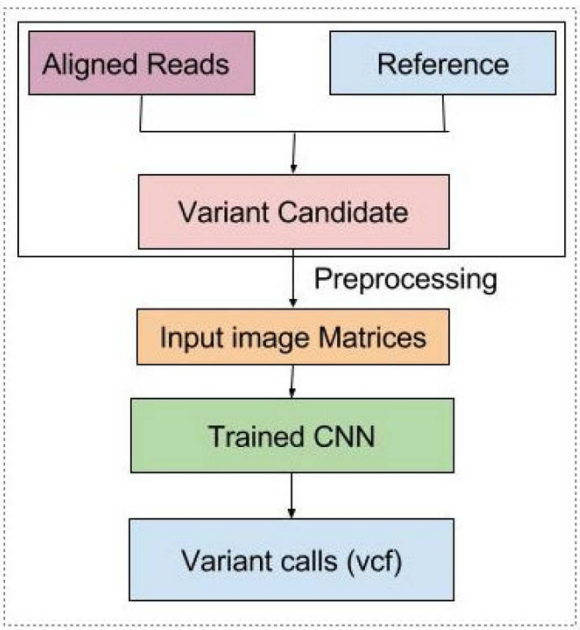


Figure 13. Proposed CNN-based Variant Caller.

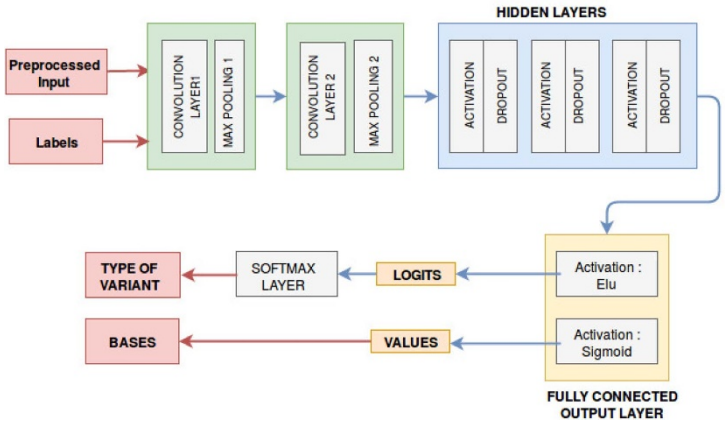


Figure 14. CNN architecture for Variant Calling.

4.2.1. Convolutional neural network

The network consists of two convolution layers with max pool layers and three densely connected hidden layers. In the convolution layers, kernels of size (2,4) and of size (3,4) are applied to generate 48 convolution filters. Three densely connected layers use exponential linear activation function [36] with a dropout rate of 0.05. The above network is trained using Adam Optimizer with learning rate 0.000 5. The output layer contains two groups of output; for the first 4 output units, we learn about the possible bases of the site of interests. For example, if the data indicates the site has a base 'C', we like to train the network to output [0, 1, 0, 0]. If a site has heterozygous variants, for example 'A/G', then we would like to output [0.5, 0, 0.5, 0]. We use mean square loss for these 4 units. For the second group of outputs, the units consist of variant type. We use a vector of four elements to encode all possible scenarios. A variant call can be of any category as mentioned above. We use a soft max layer and use cross-entropy as the loss function for these four units. The architecture of CNN is depicted in figure 14.

Training: CNN was trained on data as mentioned above. Training images were sent as a batch of size 500, each image of dimensions 15*4*3. The first 30 000 images were used for training and the next 30 000 were used for validation. Number of epochs for training was 3000. The time taken to train network on 30 000 reads of *NA12 878-chr21* was 284.289 seconds.

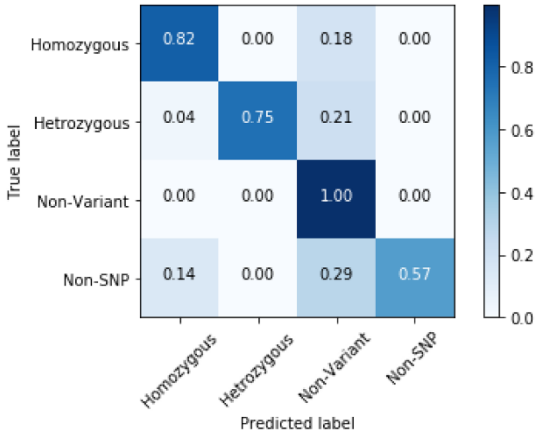


Figure 15. Confusion Matrix of Genotype Prediction.

Table 8. Genotype Prediction Report.

	Precision	Recall	f1-score	Support
Homozygous	0.75	0.82	0.78	11
Hetrozygous	0.91	0.75	0.82	28
Non-Variant	1.00	1.00	1.00	3954
Non-SNP	0.8	0.57	0.67	7
avg/total	1.00	1.00	1.00	4000

Table 9. System Specification.

Processor	Memory	Hard Disk	GPU
Intel i7-4770 CPU @ 3.40GHz	7.7 GB	1.5 TB	None

Table 10. Variant Caller Dataset.

Dataset	Species	NGS	Reference
hg38.NA12878 chr21-14069662-46411975	Homo sapiens	PacBio RSII	hg38.chr21
hg38.NA12878 chr22-18924717-49973797	Homo sapiens	PacBio RSII	hg38.chr22
NA12878 chr20.10_10p1mb	Homo sapiens	Illumina	ucsc.hg19 chr20 [38]
SRR1610046_1	Ecoli	Illumina	ecoli-str. K-12 substr. MG1655
SRR1610046_2	Ecoli	Illumina	ecoli-str. K-12 substr. MG1655
SRR1610047_1	Ecoli	Illumina	ecoli-str. K-12 substr. MG1655
SRR1610052_1	Ecoli	Illumina	ecoli-str. K-12 substr. MG1655

4.2.2. Evaluation.

To evaluate the performance of our model we have generated a confusion matrix for genotypes of variants and report of our prediction is as shown in figure 15 and table 8. It is clear from the prediction results that our model output has high specificity and moderate sensitivity.

4.2.3. Experimental setup and results

CNN is modeled using *Tensorflow* library [37] on Ubuntu 16.04 operating system, which helps in scaling the CNN network on both CPU and GPU-based architecture. For benchmarking GATK performance *GATK 3.7* with *JAVA 8* is used. The hardware specification of the system used for benchmarking is as shown in table 9.

We benchmark the performance of our variant caller network considering real execution time (in seconds) and accuracy of prediction. We have carried out a comparative study of the proposed model against GATK 3.7 Haplotype Caller for "Illumina" dataset. To prove that deep learning-based variant caller can be used across different NGS technologies, we have also done performance and sensitivity analysis on "PacBio RSII" dataset.

We have used the dataset as mentioned in table 10.

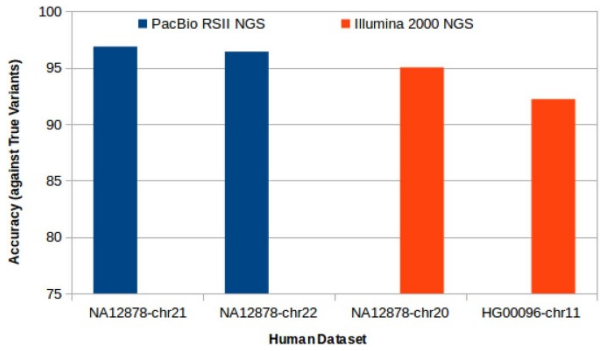


Figure 16. Prediction accuracy of Variant Caller Across NGS Technologies.

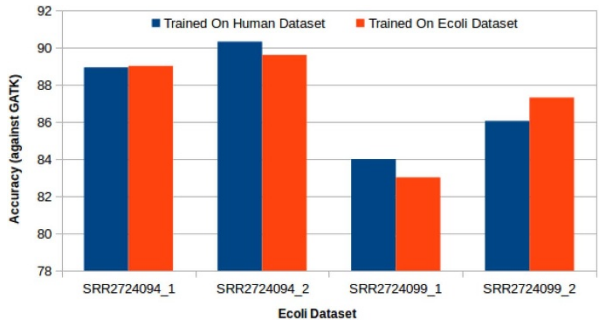


Figure 17. Prediction Accuracy if trained with same species dataset vs different species dataset.

For human data we have used four different datasets for benchmarking; *GRCh3718* [39] data is used as the reference genome. The network is trained using the first 30 000 images of human GIAB Data [35] *NA12878-chr21* dataset and validated with the next 40 000 images. The trained network is used for variant and indel prediction of *NA12878-chr22* and *NA12878-chr20* datasets. For non-human data, we have used *E. coli* data. The model is trained using *SRR1610046_1* EColi dataset [32], and tested with four other *E. coli* datasets (table 10). Timing analysis for *NA12 878-chr21*, *NA12878-chr22* and *SRR1610046_1* datasets is shown in figure 18. Prediction accuracy of the CNN variant caller when trained and tested on the same species varies between 86%–93% compared to the true variant as shown in figure 16. To observe accuracy across species, we have used variant caller trained on Human Genome *NA12878-chr21* and have used it for predicting variant in *E. coli* dataset *SRR1610046_1*. We have compared the prediction accuracy when trained on a dataset from the same species to a dataset of different species. Results are comparable as shown in figure 17.

Moreover, we tried to analyze how the model trained on one NGS technology can be used across other technologies. In this test we have used variant caller trained on *NA12878-chr21* of PacBio RSII data and predicted accuracy on PacBio RSII dataset of human *NA12 878-chr21* and *NA12878-chr22* and on Ilumina 2000NGS human dataset *NA12878-chr20* and *HG00096-chr11*. The variant prediction accuracy in both test cases is above 90% as shown in figure 16. From above experiments it is clear that variant caller trained on dataset of one species can be used for variant prediction of any other species. This will allow a non-human genome sequencing project to be benefited by depth of human truth data [40]. Moreover a model trained on one NGS technology can be used across various other NGS technologies, thus moving away from an expert driven model to a more generic data-driven models.

5. Conclusion and future work

In this paper we have explored various DNN-based models to tackle two problems in genome sequencing, namely, global/local alignment of raw NGS data and SNV identification and its classification. Proposed solutions were tested and their timing performances were benchmarked against existing de facto standard tool sets. We have observed that performance of our alignment model is comparable to BLAST but is slower than BWA-mem. Our DNN-based SNV model at the preliminary level was found to perform faster than GATK 3.7 and is highly sensitive and moderatly specefic. There can be a further boost in performance of

Table 11. CNN Variant Caller vs GATK Haplotype Caller.

Dataset	Number of Reads	Variant Caller Preprocessing Time(s)	Variant Caller Prediction Time(s)	Variant Caller Total Time	GATK Haplotype Caller Time(s)
NA12 878-chr20	52 053	12.17	0.75	12.92	20.2
SRR2724 094_1	513 785	21.67	4.05	26.576	1745
SRR2724 094_2	341 788	21.67	4.139	25.809	1716
SRR2724 099_1	1535 766	90.52	16.89	107.41	2830
SRR2724 099_2	1527 685	65.611	10.64	76.251	3322

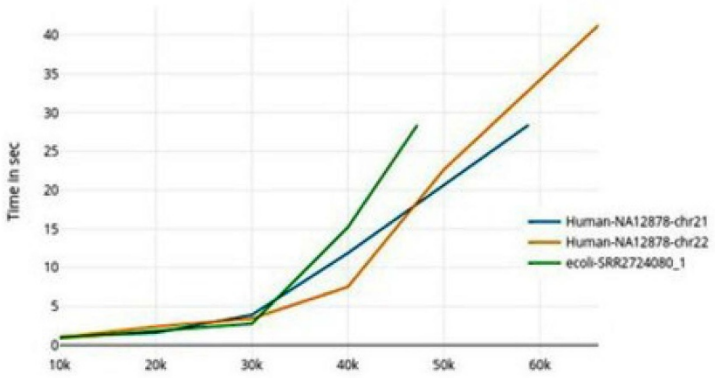


Figure 18. Variant Caller Timing Analysis.

these DNN models by using more biological information and by using multi-process and multi-threaded programming models. We have also established that use of DNN makes our model independent of underlying NGS technologies and model trained for single NGS technologies can be used across all others. Thus with help of DNN we can generate models which provide faster, reliable and highly specific solutions for identification of SNV from raw NGS reads, and will pave the way for data-centric solutions as compared to expert-based methods in the field of bio-informatics.

6. Data availability statement

The datasets used in this study are openly available at DOI or in the following link.

Data: https://github.com/gguptaiitd/genomeData

The code which is developed as part of this study is available at the following links:

Deep Conflation : https://github.com/gguptaiitd/DeepConflation

RNN : https://github.com/gguptaiitd/RNN

CoDeepNEAT : https://github.com/gguptaiitd/NEAT

A. Algorithms

Algorithm 2 Data Generation.

function QUERYGENERATION(refSubString, errorPercentage, m) ▷ Function to generate m query strings
  randomly generate Location in reference sub string
  for each random location **do**
    maxErrorLength ←Length Of Reference Sub String * errorPercentage
    errorLength ← *random*(*maxErrorLength*)
    generate Query substring of errorLength
    modify reference substring by query substring
  return array of query strings
function GenerateUnmatchedString(refSubString, n) ▷ Function to generate n unmatched string
  while $n \neq 0$ **do**
    Generate random reference String of length l
  return array of unmatched strings

Algorithm 2 (Continued)

```
procedure DataGeneration(referenceGenome, errorPercentage)
    Read Reference Genome Fasta File
    for n number of Reference Strings do                    ▷ Generate data for N reference strings
      Create Reference Substring of Length L
      for Each Reference Substring do
        QueryStrings ← QueryGeneration(RefSubStr, errorPercent, m)        ▷ create m query strings
        UnmatchedStrings ← GenerateUnmatchedString(RefSubStr, m + 50)     ▷ create m+50 unmatched strings
        for Each QueryString do
          Create Target Vector [1,0]                        ▷ 1:Match,0:Unmatched wrt reference
        for Each UnmatchedString do
          Create Target Vector [0,1]                        ▷ 1:Match,0:Unmatched wrt reference
    X ← random(QueryStrings, UnmatchedStrings)
    Y ← random(targetQueryVector, targetUnmatchedVector)
    return X, Y
```

Algorithm 3 Global Alignment Using Deep Conflation Model.

```
procedure GlobalAlignment(queryRead,referenceSequenceSet)
                ▷ Align query to reference sequences and return best matched location
  kmerSet ← [ ]
  queryLength ← lengthofSeq(queryRead)                      ▷ Get Length of query Reads
  for each refseq in referenceSequenceSet do
    while i ≠ (len(refseq) − queryLength ) do
      refKmer ← refseq[i : (i + queryLength)]
      kmerSet.append(refKmer)
      i ← i + 1
  index ← DeepConflationModel(queryRead, kmerSet)           ▷ Model match query with
  set of kmers and return highest matched index
  return index
```

Algorithm 4 Population Evolution.

```
function generateMutateChild(P,population)                  ▷ Generate Population offspirng by Mutation
  numContestant ← 2
  selContestant ← random.sample(population, numContestant)  ▷ Uniformly Sample two chromosomes
from Population
  rank ← fitnessTest(selContestant)()
  selContestantsort(selContest, rank)()
  pick ← random.choice(selContestant, P)()
  Offspring ← mutate(pick)()
  return Offspring
procedure Evolutionpopulation,p,n,q                         ▷ Evolve Population for n generations
  Children ← []
  Initialize population of chromosomes with random values
  while n ≠ 0 do
    for each chromosome in population do                    ▷ Selection of Chromosomes
      Train chromosome for 2 epochs and perform validation test
      Perform Fitness Test on Chromosome
    elites ← selectTopThree(chromosomes)                    ▷ Top 3 DNN's With highest accuracy
    Append elites to Children
    for rank of elites do
      probSum+ = p ∗ (1 − p) ∗ ∗rank
    P ← 1 − probSum
    while len(Children) < len(popluation) do
      mutateChild ← generateMutateChildP, population
      Append mutateChild to Children
```

Algorithm 5 Local Alignment Using RNN Model.

function alignmentScore(queryRead,referenceSet)
▷ Calculate Alignment Score by % of matched base pairs
scores ← []
for each sequence in reference **do**
matchScore ←get number of matched Base Pairs
scores.append($\frac{matchScore}{len(queryRead)}$)
return *scores*
procedure LocalAlignmentqueryRead,referenceSequenceSet,kmerLength,batchSize
▷ Align query to reference sequences and return best matched location
queryWords ← []
queryLength ← *lengthofSequencequeryRead* ▷ Get Length of query Reads
while $i \neq (queryLength - kmerLength)$ **do**
$queryKmer \leftarrow queryRead[i : (i + kmerLength)]$
queryWords.append(queryKmer)
$i \leftarrow i + 1$
refkmerSet ← []
for each refseq in referenceSequenceSet **do**
while $j \neq (len(rerfseq) - kmerLength)$**do**
$refKmer \leftarrow refseq[j : (j + queryLength)]$
refkmerSet.append(refKmer)
$j \leftarrow j + 1$
wordlocation ← []
wordScore ← []
for each word in queryWords **do**
refSeq ← []
while $i \leqslant \frac{refLength}{batchSize}$**do**
$refWords \leftarrow refkmerSet[i * batchSize : ((i + 1) * batchSize)]$
refSeq.append(refWords)
$i \leftarrow i + 1$
index ← *rnnAlignmentModelword*, *refSeq*
▷ RNN model return index if matched or -1
if $loc \neq -1$ **then**
$loc \leftarrow i * batchSize + index$
wordlocation.append(loc)
$refSet \leftarrow referenceSequence[loc : loc + queryLength]$
score ← *alignmentScorequeryRead*, *refSet* ▷ Calculate alignment score of query Read and Reference Sequence
wordScore.append(score)
location ←Best Score Location
return *location*

B. Experiment details of deep conflation model

B.1. Simulated data generation

Simulated data is generated by adding random errors in the form of substitution and indels of bases of corresponding reference string. Through this scheme, we intend to generate data which has close resemblance with actual reads obtained from the NGS sequencer. Also, by using error as a parameter in the data generator, we can generate and experiment with datasets containing varying degree of errors. This will help in establishing correlation between the percentage of error in reads and the prediction accuracy of the neural network, if any exists. The algorithm for data generation is described in Algorithm 2 (appendix A).

The output of data generation is jumbled using uniform random distribution. For experimentation, three data sets of inputs and targets are generated - namely, training dataset, validation dataset and test dataset. Each dataset is generated by the method as described above, but the number of matched and unmatched corpuses are different due to uniform random sampling of batch data. Data is generated for 4000 reference strings, total database of corpuses is of the order 61*4000. The dataset is split into 80, 20 ratio for training and validation set. For test, another 1000 reference string dataset is generated. The length of each reference string is varied from 15–170 bps for experimentation. Details of DNN models which use this dataset are discussed in following section.

B.2. Deep conflation model for alignment

Model overview

Given a read k-mer 'x', the model ranks a set of reference k-mers $\langle r_1, r_2, ..., r_n \rangle$, so that the reference k-mer most similar to the given read gets the highest rank. The k-mers are first preprocessed and encoded as per the following scheme:

A : 0, C : 1, G : 3, T : 4.

The model consists of two parts:

1 Extracting finite dimension feature vectors from the encoded read and reference k-mers through CNN
2 Ranking the reference features based on cosine similarity with read features. $\langle r_1, r_2, ..., r_n \rangle$ are ranked in order of decreasing similarity.

CNN feature extractor

Let 'x' be a string containing 'k' characters. Then, character encoding of x is obtained as $x = [x_1, x_2 ..., x_k]$. x is then fed as input to the CNN, and convolution is applied using three different filters of sizes 2, 3 and 4 to extract features corresponding to bigrams, trigrams and 4-grams. The 't' th output of a convolution with filter size 'f' is obtained as follows: Where W_f is the convolution weight, b_f is the bias and $s_{t:t+f-1}$ is the vector obtained by concatenating s_t to s_{t+f-1} Feature map of convolution with filter size 'f' can be defined as: Max pooling is applied to feature map h_f to obtain $\hat{h_f}$. Feature maps obtained with filters of sizes 2, 3 and 4 are concatenated to form a vector representing the entire input string/character sequence $y = [\hat{h_2}, \hat{h_3}, \hat{h_4}]$. This final feature vector is fed to the ranker module. The process of feature extraction is repeated for each read k-mer and reference k-mer.

$$h_{ft} = tanh(W_f\, s_{t:t+f-1} + b_f) \tag{B1}$$

$$h_f = [h_{f1}, h_{f2}, ..., h_{f(k-c+1)}] \tag{B2}$$

Ranker

After obtaining the feature vector y_x for read k-mer x, and set of feature vectors $y_{r_i} =< y_{r_1}, y_{r_2}, ..., y_{r_n} >, 1 < i < n$ for reference k-mers $\langle r_1, r_2, ..., r_n \rangle$ the ranker computes cosine similarity between y_x and each of y_{r_i} as follows. Reference k-mers are ranked by these scores with respect to a given read k-mer, with the best matching reference k-mer awarded the highest rank.

$$R(x, r_i) = \frac{y_x^T y_{r_i}}{||y_x||.||y_{r_i}||} \tag{B3}$$

Training

For training, we have used simulated data as described in the previous subsection. We feed the model a batch of 100 such *(*reference, mutant*)* pairs, wherein the model is expected to learn to match the mutated string to its original reference string, and assign it the highest rank. Using the similarity scores obtained from equation (B3), the posterior probability of correct reference string, given the mutant string is defined as follows: Where r^+ is the correct reference string to which x should match. γ is a hyper-parameter set to 10. *Ref* is the set of reference strings, containing r^+ and 99 other random non-matching reference strings. While learning, the model tries to minimize the following loss function: Where θ denotes the model parameters to be learnt, and product is over all examples in a batch. Adam Optimizer with learning rate of 0.000 2 is used for loss minimization.

$$P(r^+|x) = \frac{\exp(\gamma.R(x|r^+)}{\sum_{r' \in Ref} \exp(\gamma.R(x|r'))} \tag{B4}$$

$$L(\theta) = -\log \prod_{x, Ref} P(r^+|Ref) \tag{B5}$$

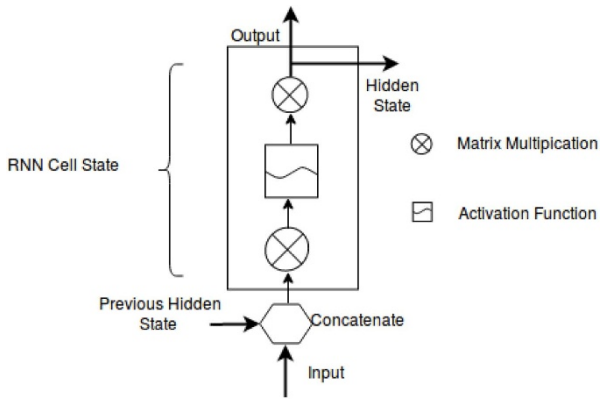


Figure C1. Vanilla RNN Cell Design.

Testing

During testing, a read k-mer is fed to the model and features extracted from this input are compared with feature vectors obtained from reference k-mers. Thereafter, reference k-mers are ranked in order of similarity (based on equation (B3)), and the highest rank reference k-mer is output as the most probable match for the given read k-mer.

C. Design and training of RNN models

C.1. Vanilla RNN

A Vanilla RNN is designed to include an additional context layer, where the activation of hidden units from previous feed-forward process is stored [41]. These hidden state values are concatenated with input which generates a three-dimensional vector of shape "$sequence\_length, batch\_size, hidden\_size + input\_size$. These vectors are then fed to the RNN cell with fully connected nodes. Each cell of RNN stores value as hidden state which is of dimension "$batch\_size, hidden\_size$''. The output of the RNN cell is then fed to the FC layer while the hidden state is fed to the input layer of the next cell. The design of a Vanilla RNN cell is as shown in figure C1.

C.1.1. Training

The network was trained on *simulated data* for 50 epochs, with 380 batches of size 10 (number of batches = 61*4000/batch size/sequence length) for each epoch. For each epoch, at an interval of 100 batches, training loss and training accuracy of predicted output was calculated. To observe overfitting, a validation test was performed on validation data at the same interval. With learning rate of 0.001 for optimizer the plots of loss function, training accuracy and validation accuracy is shown in figure C2. The value of hyper-parameter for Vanilla RNN is as in table 2.

C.2. LSTM

In order to model global alignment of genome sequences by comparing reads of long lengths, we have explored LSTM cell in our Recurrent Neural Network architecture. Benefit of LSTM is that it helps to avoid long-term dependencies and thus remembers data patterns for long time(sequences). From many different LSTM configurations available, we have used Hochreiter & Schmidhuber LSTM [42]. A cell of LSTM consists of hidden state and cell state. *Hidden state* of LSTM cell is used for modifying state of next LSTM cell, while *Cell State* dictates output of current cell. Both values for initial LSTM cell are initialized to zero. Similar to RNN, input to cell is concatenated with previous hidden state before being fed to next LSTM cell.

C.2.1. Training

The LSTM network was trained for 100 epochs with the number of batches and batch size the same as that of the RNN model. Training set and validation set are also kept same as that of RNN model. Hyper-paramenters of the LSTM network are show in table 2. Unlike RNN, the training loss and training accuracy in the LSTM model are calculated for each epoch, while the validation test is done at an interval of 100 batches for each epoch. The training loss, training accuracy and validation accuracy of the LSTM model is shown in figure C3.

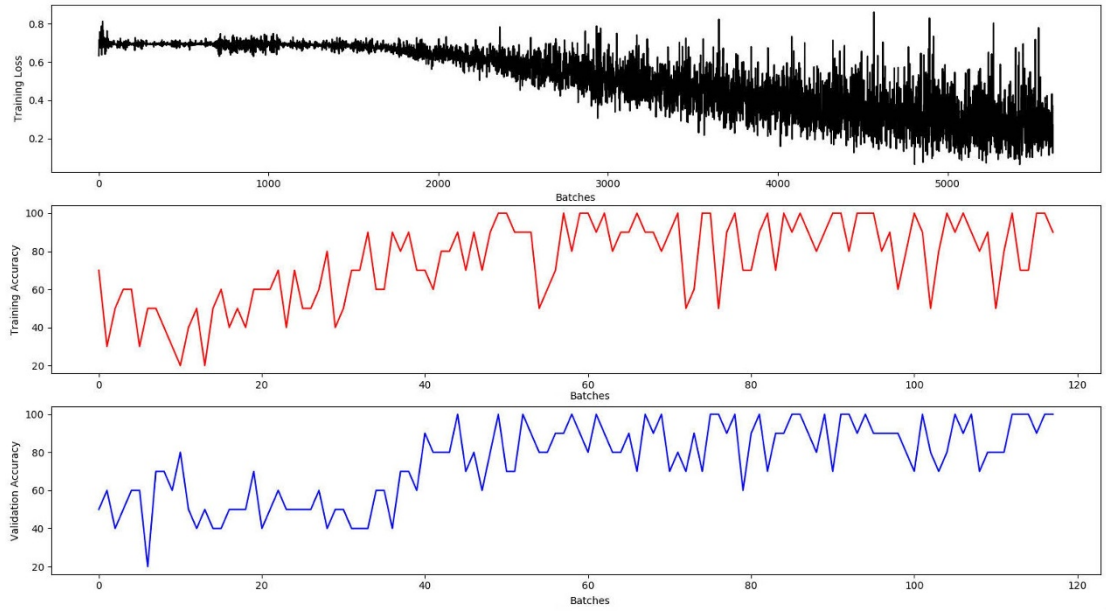


Figure C2. Training Loss, Training and Validation Accuracy Of Vanilla RNN.

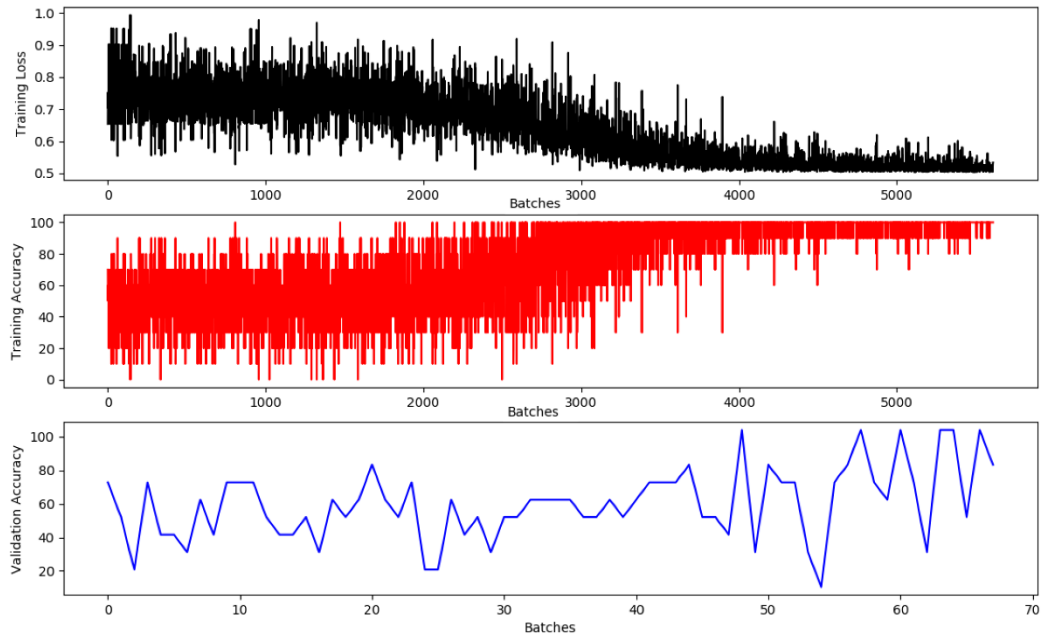


Figure C3. Training Loss, Training and Validation Accuracy Of LSTM.

D. Data preprocessing for CNNbased variant caller

D.1. Data preprocessing

This process involves the generation of variant candidates and generation of labeled data to train CNN. For variant generation, the aligned BAM file is read using Samtools and corresponding CIGAR [43] strings are decoded. From the CIGAR string, different alleles along with their position in reference reads are identified by comparing it with reference read. These alleles are then classified either as reference-matching base, reference-mismatching base, an insertion or as a deletion [17].

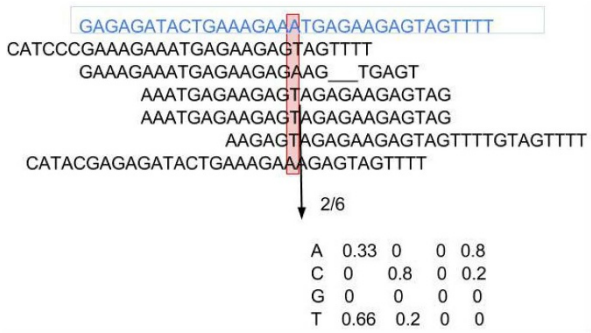


Figure D4. Position Specific Frequency Matrix.

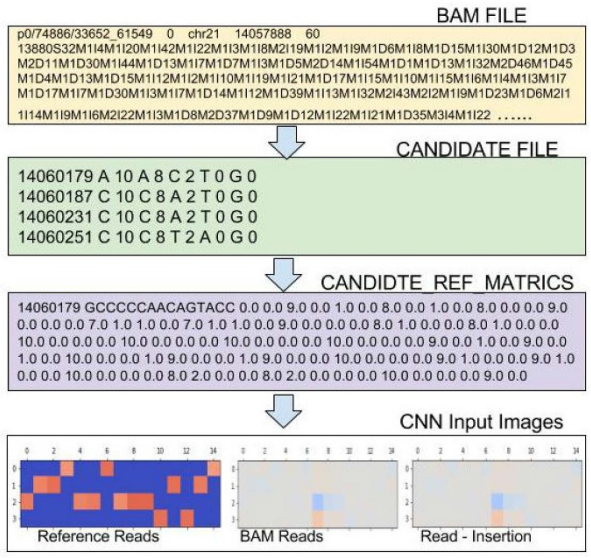


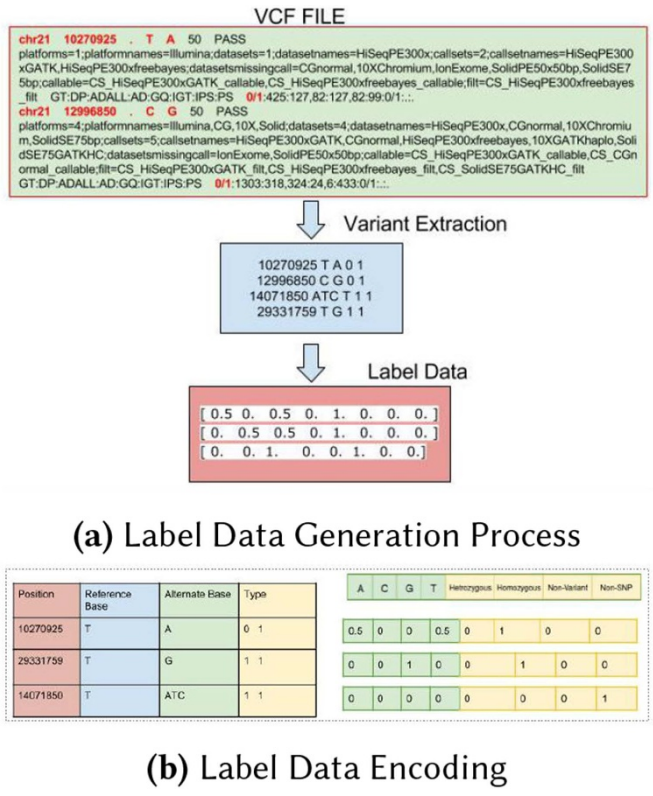
Figure D5. Data Transformation during Preprocessing

The frequency of each distinct allele for a position is calculated using position-specific frequency matrices (PSFM) [44] as shown in figure D4. If the frequency is above the set threshold, the allele site is considered as a variant candidate.

For all possible variant candidates, the variant string is generated by reading the fixed length of bases from reads, with the variant site as median. These variant candidate strings are encoded into three matrices. The first matrix is created by encoding the expected reference sequence using one-hot-like encoding. It encodes a number of reads that are aligned to a reference position. The second matrix encodes the difference of all the bases observed in the read-reference alignment. Reads with 'N' nucleotides are ignored. The third matrix is similar to the second matrix, except none of the insertion bases in the reads is counted. All the matrices are input to CNN for both training and variant calling. Figure D5 depicts data transformation at each stage of the data preprocessing.

Labeled data: labeled data is created by reading the corresponding variant file in VCF format. Only those true variants are considered for labeling which fall in high-confidence regions of the reference genome. For all true variants, their position, reference base, alternate base and their genotypes are read. These variants and their genotypes are then encoded as shown in figure LabelDataEncoding.

The process of generating label data and data transformation for labeling is shown in figure LabelDataGeneration.



Label Data Generation.

ORCID iD

G Gupta https://orcid.org/0000-0002-6459-8203

References

[1] Metzker M L 2010 Sequencing technologies-the next generation *Nat. Rev. Genet.* **11** 31–46
[2] Church G M 2005 The personal genome project *Mol. Syst. Biol.* **1** 1
[3] Bamshad M J, Ng S B, Bigham A W, Tabor H K, Emond M J, Nickerson D A and Shendure J 2011 Exome sequencing as a tool for Mendelian disease gene discovery *Nat. Rev. Genet.* **12** 745–55
[4] Mielczarek M and Szyda J 2016 Review of alignment and SNP next-generation sequencing data *J. Appl. Genet.* **57** 71–9
[5] El-Metwally S, Hamza T, Zakaria M and Helmy M 2013 Next-generation sequence assembly: four stages of data processing and computational challenges *PLoS Comput. Biol.* **9** e1003345
[6] Houtgast E J, Sima V-M, Bertels K, and Al-Ars Z, Computational challenges of next generation sequencing pipelines using heterogeneous systems *12th Int. Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* pp 1–4
[7] Olson N D *et al* 2015 Best practices for evaluating single nucleotide variant calling methods for microbial genomics *Front. Genet.* **6** 235
[8] DePristo M A *et al* 2011 A framework for variation discovery and genotyping using next-generation DNA sequencing data *Nat. Genet.* **43** 491–8
[9] Hwang S, Kim E, Lee I and Marcotte E M 2015 Systematic comparison of variant calling pipelines using gold standard personal exome variants *Sci. Rep.* **5** 17875
[10] Schbath S, Martin V, Zytnicki M, Fayolle J, Loux V and Gibrat J-F 2012 Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis *J. Comput. Biol.* **19** 796–813
[11] Asgari E and Mofrad M R 2015 Continuous distributed representation of biological sequences for deep proteomics and genomics *PloS One* **10** e0141287
[12] Aoki G and Sakakibara Y 2018 Convolutional neural networks for classification of alignments of non-coding RNA sequences *Bioinformatics* **34** i237–i244
[13] Ganesh P, Gupta G, Saini S and Paul K 2018 *Biorxiv* Nucl2vec : Local alignment of dna sequences using distributed vector representation (available at: https://www.biorxiv.org/content/early/2018/08/29/401851)
[14] Curnin C, Goldfeder R L, Marwaha S, Bonner D, Waggott, D, Wheeler M T and Ashley E A 2019 Machine learning-based detection of insertions and deletions in the human genome BiorXiv:10.1101/401851
[15] GATK, Gatk best practices. (available at: https://software.broadinstitute.org/gatk/best-practices/)
[16] Lawrence M 2014 Introduction to variant calling *Lecture Series* University of Bath
[17] Poplin R, Newburger D, Dijamco J, Nguyen N, Loy D, Gross S S, McLean C Y and DePristo M A 2017 Creating a universal SNP and small indel variant caller with deep neural networks *Nat. Biotechnol.* **36** 983–7
[18] Angermueller C, Pärnamaa T, Parts L and Stegle O 2016 Deep learning for computational biology *Mol. Syst. Biol.* **12** 878

[19] Tran N H, Zhang X, Xin L, Shan B and Li M 2017 De novo peptide sequencing by deep learning *Proc. Natl Acad. Sci.* **114** 8247–52
[20] Hou J, Adhikari B and Cheng J 2017 Deepsf: deep convolutional neural network for mapping protein sequences to folds arXiv:1706.01010
[21] Alipanahi B, Delong A, Weirauch M T and Frey B J 2015 Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning *Nat. Biotechnol.* **33** 831–8
[22] Lanchantin J, Singh R, Wang B and Qi Y 2017 Deep motif dashboard: Visualizing and understanding genomic sequences using deep neural networks *Pacific Symposium on Biocomputing 2017* (World Scientific) pp 254–65
[23] Zhang J M and Kamath G M 2016 Learning the language of the genome using RNNs.
[24] Mohan B A, A. and N. Chandra 2015 *Escherichia coli* str. K-12 substr. MG1655 complete genome. (available at: https://www.ncbi.nlm.nih.gov/nuccore/CP012868.1)
[25] Lander E e a, Homo sapiens chromosome 20, grch37 primary reference assembly. (available at: https://www.ncbi.nlm.nih.gov/nuccore/CM000682.1)
[26] Hattori M E A 2009 *Homo sapiens* chromosome 20, GRCh37 reference primary assembly. (available at: https://www.ncbi.nlm.nih.gov/nucleotide/CM000683.2)
[27] Lander E E A 2013 *Homo sapiens* chromosome 22, GRCh38 reference primary assembly. (available at: https://www.ncbi.nlm.nih.gov/nucleotide/CM000684.2)
[28] Gan Z, Singh P, Joshi A, He X, Chen J, Gao J and Deng L 2017 Character-level deep conflation for business data analytics, in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE Int. Conf. on* pp 2222–6 IEEE
[29] Miikkulainen R *et al* 2017 Evolving deep neural networks *Artificial Intelligence in the Age of Neural Networks and Brain Computing* ed R Kozma *et al* (Cambridge, MA: Academic Press) pp 293–312
[30] Stanley K and Miikkulainen R 2002 Evolving neural networks through augmenting topologies *Evol. Comput.* **10** 99–127
[31] Heitzinger C 2008 Mutation operator (available at: www.iue.tuwien.ac.at/phd/heitzinger/node27.html)
[32] Ecoli read data 2020 (available at: https://github.com/gguptaiitd/genomeData)
[33] JiaShun-Xiao python-implement-fast-blast-basic-local-alignment-search-tool (available at: https://github.com/JiaShun-Xiao/python-implement-fast-BLAST-Basic-Local-Alignment-Search-Tool)
[34] Zook J M, Chapman B A, Wang J, Mittelman D, Hofmann O M, Hide W and Salit M 2018 Integrating human sequence data sets provides a resource of benchmark snp and indel genotype calls *Nat. Biotechnol.* **3** 246–51
[35] Na12878 data 2014 (available at: https://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/release/NA12878_HG001/NISTv3.2.1/)
[36] Clevert D-A, Unterthiner T and Hochreiter S 2015 Fast and accurate deep network learning by exponential linear units (ELUs) arXiv:1511.07289
[37] Abadi M *et al* TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, software available from tensorflow.org (available at: https://www.tensorflow.org/)
[38] *Homo sapiens* chromosome 20 HG10 reference assembly (available at: http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/chr20.fa.gz)
[39] GATK genome reference data (available at: https://github.com/bahlolab/bioinfotools/blob/master/GATK/resource_bundle.md)
[40] Nielsen R, Paul J S, Albrechtsen A and Song Y S 2011 Genotype and SNP calling from next-generation sequencing data *Nat. Rev. Genet.* **12** 443
[41] Elman J L 1990 Finding structure in time *Cogn. Sci.* **14** 179–211
[42] Hochreiter S and Schmidhuber J 1997 Long short-term memory *Neural Comput.* **9** 1735–80
[43] McKenna A *et al* 2010 The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data *Genome Res.* **20** 1297–303
[44] Libbrecht M W and Noble W S 2015 Machine learning in genetics and genomics *Nat. Rev. Genet.* **16** 321