Variant Classification Using Convolution Neural Network

Ron Kandelshein

I. ABSTRACT

The following paper will review the attempt to create a method to classify a variant out of a viral sample quickly and efficiently. The ability to so with high accuracy can contribute to the analysis and prediction of a viral pandemic and its origin. To tackle this classification problem, I choose the 1D Convolution network design approach. As I intended to represent the data as a 1D vector, using a 1-D convolutional design can be faster than using other more complex designs and can compensate the large number of classes that should be classified. The paper will explain and review the methods of use and display the results of the effort.

II. INTRODUCTION

Identification and classification of viruses are essential to avoid an outbreak like COVID-19. Regardless, the feature selection process remains the most challenging aspect of the issue. The most commonly used representations worsen the case of high dimensionality, and sequences lack explicit features.

The COVID-19 family has been characterized as a (+) ssRNA (positive-sense single-stranded RNA virus) and has been identified in avian and mammalian hosts (including humans). This virus family has a genome length that ranges between 26 and 32 k base pairs.
To address the challenges associated with identifying and classifying all species, several techniques have been proposed for the analysis and comparison of genomic sequences.

In the following paper I will cover the process of the work to classify genomic sequence to a linage using a CNN network. The work was implemented via python language using different libraries to assist in data representation and network modeling and evaluation.

III. METHOD AND DESIGN

On this section I will review the process design in the order that it was executed in the code.

A. Collecting the Data

The conducted experiments used a database of 25,000 samples, acquired from https://www.covid19dataportal.org. In-order to use the data, 2 files were downloaded:

1. raw data - Genomic data in FASTA format
2. label data - Linage to accession id mapping in CSV format.

B. Data Pre-Processing

To pre-process the data, I used numpy and pandas libraries in python. To extract the variant sequence from the raw data I used the structure of the text in the FASTA file. Before each sequence, the accession ID of it appeared in the line above it in specific structure. By extracting all of the ID's from the Label data, using pandas python library, I iterated on the values and matched each variant sequence to its ID.

The next step was to match each accession ID to its corresponding Linage by creating a mapping from the label data by using python dictionary functionality and store the mapping of it. After doing so, the Genomic data was represented a list of lists:

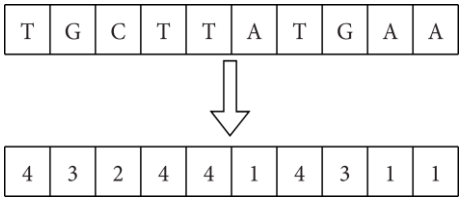
- Each Inner list represents a full sequence
- Each element of the inner list represents a DNA base.

To label each sequence for the training, I have mapped each accession ID to its linage via pandas library. By doing so, I was able to map every sequence to it's linage according to the common accession ID in both data structures.

The linage mapping is stored in a corresponding list, where each element represents a linage of the sequence represented in the sub-lists by matching the list indices

Preprocessing data is the most critical step in most machine learning and deep learning algorithms that involve numerical rather than categorical data. The genomic sequence in the DNA dataset is categorical. The encoding technique is the process of converting the categorical data of DNA bases into numerical form.

In Label encoding, each DNA Base in the sequence is assigned an index value: A-1, C-2, G-3, and T-4.



In order to initiate the learning process, I manipulated the Genome data into numeric sequence as displayed above – meaning the feature of genome is represented by 1D vector. As the network needs all the inputs to be in the same size, I added padding to the vector if it's length was shorter than the maximum length.

The label data was also transformed to 1D vector for the same reason. To each sample vector a corresponding label vector was created, where the index of it's linage got the value of 1 and all the other elements were nullified. This resulted in 25,000 label vectors in length of the number of linages to classify (391 on my case)
To sum up, the data is represented as follows:

- Genome Sequences:
  Each sequence is represented by a 1D vector in length of the sequence. There are a total of 25,000 vectors.
- Linage Mapping:
  Each sequence is labeled by a zero vector in length of 391, where the index of its linage it set to the value 1.

Partial Sample spread of the data example:

count	**Linage ID**	count	**Linage ID**	count	**Linage ID**
24	**180**	205	**345**	5257	**368**
24	**72**	189	**319**	2618	**114**
23	**4**	184	**26**	2608	**25**
23	**256**	183	**74**	2169	**24**
23	**162**	162	**164**	1473	**320**
23	**294**	144	**88**	849	**16**
23	**18**	143	**27**	548	**136**
22	**252**	139	**32**	420	**308**
22	**17**	135	**278**	382	**152**
21	**231**	133	**92**	361	**50**
21	**139**	132	**247**	336	**23**
21	**177**	115	**334**	242	**199**
21	**60**	114	**301**	237	**86**
20	**343**	114	**127**	217	**215**

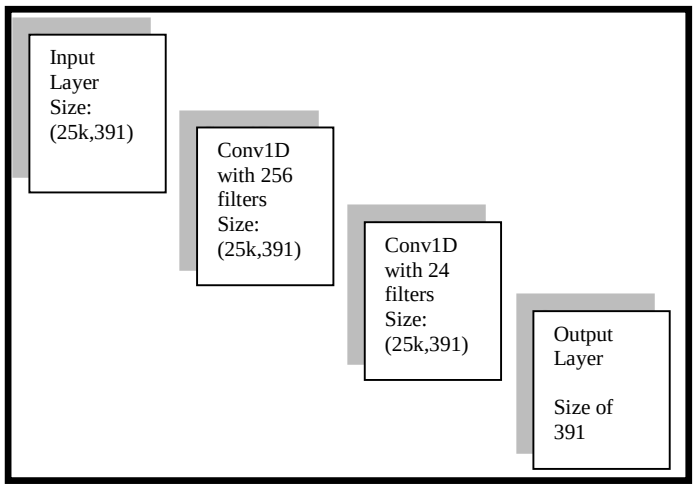
C. Network Design

The network was implemented in python via TensorFlow and Keras libraries. I tried several different network designs which were based on the concept of CNN network calcification.

During the design process I encountered that adding to many layers to the design architecture caused the network to over fit – which was expected due to large amount of parameters in the network and the small data set. In order to prevent overfitting, I tried explore different methods such as:

- Batch normalization
  Performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.
- Max pooling
- The purpose of the pooling layers is to reduce the dimensions of the hidden layer by combining the outputs of neuron clusters at the previous layer into a single neuron in the next layer
- Increasing Dropout rate
  Reduce the chance to over fit by nullifying the contribution of some neurons towards the next layer and leaves unmodified all others
- Flatten data
  Adjusting the input from the CNN layer into the the fully connected layers

After several trial and error, the final design of the network is as follows:



Between the two Conv1D layers I have added max pooling (pool size=2) and dropout of 20%. Following by adding between the 2nd Conv1D layer and the output layer a similar pooling as before, batch normalization and data flatten.

IV. Evaluation Methodology

In this section I will display the training & result evaluation process that was done.

A. *Data Sets*

For the experiment, sub-datasets were created:

- Training Set:
  20,000 samples were used to training the model with spread of 80% training samples and 20% validation samples.
- Testing Set:
  5,000 samples were used to evaluate the model after the training.

B. *Training the network*

The data was trained in several iterations in the following parameters:

- Epochs: 8
- Batch Size: 12
- Batch Shuffle: True
- Optimizer: 'Adam'
- Loss functions: categorical cross entropy
  Categorical Cross Entropy is also known as Softmax Loss. It's a softmax activation plus a Cross-Entropy loss used for multiclass classification. Using this loss, we can train a Convolutional Neural Network to output a probability over the N classes.

$$CE = -log\left(\frac{e^{s_p}}{\Sigma_j^C e^{s_j}}\right)$$

After the first results, I have saved the model in a TensorFlow dedicated format to have the ability to continue the training process for additional epoch to increase its accuracy and results.
For each epoch, if the loss function value was reduced save was triggered. The total amount of times a have trained the network was 4 iterations – meaning 24 epochs.

The final model test parameters that I got was:

- Train accuracy: 74%
- Validation Set accuracy: 63%
- Training Loss: 1.0442
- Validation Loss: 2.6821

I have noticed that the in each iteration these parameters percentage increases by a 3-4%.

V. Results

A. *Result Overview*

To evaluate the model, I have predicted the test set labels on it and evaluated the accuracy. The accuracy that I was 68%. Meaning out 5,000 sequences the model predicted the correct linage of 3,400 of them.

B. *Result Discussion*

As shown in section 3.B, the data spread is no equal by a large margin. Only 71 linages out of 391 has above 40 samples of data. That indicated that a large number of epochs will be needed to fully train the samples on the network. More-over, I did not split evenly the test and the train datasets which may cause some linages sequences to not be trained by the model at all.

VI. Conclusion

The paper has covered my attempt to classify variant sequences to its linage using Machine leaning. In the process and due to the results I have learned the large amount of data that is required to create a good model and the impotence of representing the data in a way that will minimize the network parameters to train.

The conclusions from my work so fat arte as follow:
Improve design: As we are dealing with sequential data, there are multiple existing models that can be explored and improve the model accuracy other than the one I choose.
Change input size: As each genome is a very long sequence, the data shape can be minimized to ease the learning process – i.e., Kfold method. Representing short DNA Base sequences in a numeric value will greatly reduce the input size
More Samples: Can retrieve the full samples if enough processing power at hand – more data will give the model a bigger training set and improve the accuracy. I wasn't able to train any bigger dataset.
Class representation: Not all Linages has the same representation in the data set – better results would be archived if in the pre-processing stage I will adjust the data to have even-representation of the linages.

VII. PROJECT FILES

1. Full data spread: label_spread.xlsx
2. Execution run file: main.py
3. Helper method: Helper.py