A Deep Learning Approach to Variation Detection in Genomics

Ayush Nayak
Westview High School, San Diego California
(No Mentor)

January 2021

Abstract

Many diseases that are not pathogenic are grounded in genetics. Discovering which genes cause these diseases is a modern challenge, made easier with the convergence of both widely available public data, in addition to the advent and widespread use of deep learning. Deep Learning models are able to classify and make predictions with very high degrees of accuracy, almost 95% in this study. With such high levels of accuracies, if trained properly these models should have some insight that allows them to get that level of accuracy. Most of this study is dedicated to first training a model to high accuracy on the chosen target of Adenomas and Adenocarcinomas, before using methods to glean this insight and realize the most important genes identified by the model itself. In the end the model was able to identify many new combinations of genes that had a strong correlation with Adenomas as well as do it in a relatively short amount of time than most GWA studies.

Contents

List of Figures

1 Introduction

A challenge in bioinformatics and genomics remains: detecting and identifying correct sets of variations that cause or deeply correlated with a disease. Genomics studies are currently conducted by looking for variations that occur more frequently in a population with the targeted phenotype than the control group. Statistically significant variations are then pursued and fully sequenced, to figure out which variations, somatic or germline contribute to a genetically based disease.

While this method works fine for instances where a single gene causes a disease, most of the times this is untrue, with many genetically based diseases caused by a combination of multiple variations, as opposed to a single or collection. For instance certain diseases such as Sickle Cell Anemia are caused by a collection of genes instead of just one. Detecting these requires statistically looking at every pair of variations, or to target three, every 3. Taking the binomial coefficient for n=20,000 and varying k, analyzing pairs of variations would take around 199 Million operations, and analyzing all possible triplets requiring 1.3 trillion. This number only grows in practice across an entire dataset, which would have a much larger n value. Evidently this is unfeasible, and for the wide amount of variations that exist, as well as the discrepancies in data and balancing, even more issues could be introduced. Machine Learning models themselves have had a slow uptake in bioinformatics and genomics itself, and these days are being used mostly as a method for analyzing variations already known to correlate, instead of being used to find new ones.

Deep Learning itself has advantages of being somewhat automated, as well as being highly accurate in predicting and learning outcomes, as long as overfitting or balance issues are not introduced. One of the key issues with this sort of analysis is the "black box" factor, meaning researchers have no idea how the machine learns, or how it classifies, only the fact that it can classify. There are ways to get this information however, and it is possible, however time cost is high. Deep Learning is starting to blossom and gain adoption in many fields across biology, as well as the rest of science and coinciding with the arrival of large amounts of freely available variation data allows for many new discoveries and a wealth of new information on these topics. Throughout Genomics and Bioinformatics machine learning models have been applied to traditional loads such as clustering and prediction, although they have almost never been applied too identifying variants, that has only ever been done with statistical analysis. This is due to the complexity and the fact that machine learning models can rarely come up with conclusions, rather only being results based.

The idea behind this study is very simple. Machine learning models are very good at classifying data, by the end of this study with an accuracy of almost 94% over testing data. In order to make these gains, the models themselves must have some conclusions, or at least some insights into the genes and their makeup themselves, unless the model is overfitting, which is what a lot of this study focuses on, training the model. The second part is actually finding these insights, ML models are notorious for being "black boxes" but there are ways to find an importance level for each input value, and that is done near the end. This entire analysis process was done on a target and the most important genes found were manually assessed and are summarized.

In this study, I first start out by exploring the different methods and models that can be used to classify genes, before going ahead and implementing one on a full dataset, and then trying to do the hardest part of machine learning, understand why the model thinks what it does.

2 Materials and Methods

First, I will go over machine learning models, and choose which ones to use, ML models are composed of different layers all of which do certain calculations. There is an input layer, which takes in all data, as well as an output layer, which spits out the answer. The output layer normally just has one node.

Models are sets of equations, and are trained through whats called backpropagation, where the coefficients for different equations are tuned accordingly over and over again through training data, the model discerning what part of it was not accurate. Each time a model goes through a chunk of training data, it is called an Epoch. The code is in Python 3.9, although what it means is normally below.

Neural Networks work by having an input layer take in some data, and then having layers of nodes, each node is connected to a node in the next layer, and what they do is add some weight or multiply a value by some number, before feeding it foreword through the network. A classic network with nodes going from A (input) to B to C to many intermediary nodes which all reference the node behind them, followed by an output layer are called MLP or feed foreword networks.

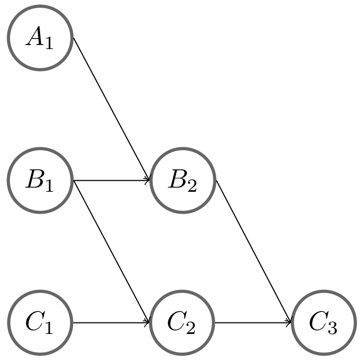

Figure 1: Nodes to explain Neural Networks and Machine Learning

This picture shows a very simple machine learning model. with 3 inputs, and 1 output. Starting with each input node, the value passes to the next node, where the two values are multiplied by a certain value and then added to a certain additional number. Training would adjust the number each value is multiplied by. Multiplying and adding can make a number very large, so through a function called an activation function, the output is mapped between 1 and 0. The one being used is a sigmoid $\frac{1}{1+e^x}$.

Mathematically, for node B_2, the equation from inputs A_1 and B_1 is as follows:

$$B_2 = sig(C_1 \cdot A_1 + C_2 \cdot B_1)$$

A_1 and B_1 are inputs, both connect to B_2, their value is multiplied by a certain amount each, (C_1 and C_2, for A_1 and B_1 respectively) and then summed for the output. By adjusting and combining these weights, an output is found for each node.

This is a very basic explanation, and only covers one type of model, the Feed Foreword or MLP model.

2.1 Model Approach

Deep Learning itself, hinges on three parts, models, data input, and data analysis. All variations were taken into a master union set, full of every unique variation present was first created. As most of these are extremely rare, any variation under 0.5% was excluded and removed from the set, creating a nice input format. For instance, for a dataset with genes A, B, C, D, E and F, for a person with no genes, the set would be [0,0,0,0,0,0]. For a person with all of them, [1,1,1,1,1,1,1]. For a person with A B and C, [1,1,1,0,0,0]. For a person with A, C and F [1,0,1,0,0,1]. TCGA and dbSNP as well as dbGAP's databases were used and conglomerated in accordance with the method shown above.

This list is a catalog, for instance with stars, for a set of Polaris, Vega and Arcturus, in positions [Polaris, Vega, Arcturus], [1,1,1] would mean all three stars are present, [0,1,0] means only Vega is present, and so on. "Position 2" would be the presence of Vega, similarly with genes, a certain gene has position 2 in the catalog, its not the "second gene" merely it has that position in the catalog, and a master catalog was kept to say which gene is at each position.

Model wise, neural networks can prove to be more or less efficient with different amounts and node composition for layers. A proof of concept stage was used to assess different combinations of nodes, and come up with the best combinations to use for the final model. PyTorch 1.7.1 is used along with of course Python Version 3.9.1 Latest at the time of writing, Python itself chosen for research friendliness, and ease of use due to the very experimental nature of the project. Before selecting a target to do statistical analysis as well as starting that analysis and gathering data, to prove the usefulness of deep learning, as well as test its implementation on smaller controlled datasets, and determine a model type, a proof of concept or pre-project was used.

2.1.1 Proof of Concept

A proof of concept in this sense, would be creating a dataset with a certain trend, artificially, and using different types of machine learning models to evaluate it, testing for accuracy, and streamlining pre data. The reason for this is to validate the efficacy of the model. Creating a model based around a certain trait, for instance having a certain phenotype only show up if a single gene is present, or from a host of factors, a model being run after training should have high levels of accuracy. In addition models once deconstructed should show that they are actually using the trait being targeted.

Starting with an example, generating a simple dataset with a tensor of boolean values for a disease, for instance: These are not actual genes, but examples labeled A-J. For instance person 1, with output "True" has the A, B, C, D, and J genes. The leftmost column denotes the output, either True or False, while the rest denote the inputs.

	A	B	C	D	E	F	G	H	I	J
True	True	True	True	True	False	False	False	False	False	True
True	False	True	False	True	True	True	True	False	True	True
True	True	True	True	True	True	True	False	True	False	True
True	False	True	False	False	False	False	True	False	False	True
False	True	False	False	True	False	True	True	True	False	False
False	True	False	False	True	False	False	True	True	True	True
False	False	False	False	True	True	False	False	True	False	True

In this test dataset, True corresponds with position 2 in the array. Model wise, using an overly complex model would not really help with the validity of this model, actually using a linear model would be valid. The model itself used is very basic, a single Linear model, with a single layer and activation function, with a variable number of inputs.

```python
class MLP(nn.Module):
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        self.layer = nn.Linear(n_inputs, 1)
        self.activation = nn.Sigmoid()

    def forward(self, x):
        return self.activation(self.layer(x))
model = MLP(1024)

criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(),
                      lr=0.01, momentum=0.9)
```

Note: Technically a single layer model is NOT an MLP (Multi Layer Perceptron model, but the nomenclature was used anyways, as the model was built up later and function names are trivial.)

The dataset was generated with the correct "gene" at position 2, like in the example above, however with a length of 1024, and a total dataset size of 2048 items. Plotted below are the results over many epochs versus accuracy.

As expected the model was very fast to identify and accuracy jumped quite quickly. The reason there was a bit of training to happen around 30-50 epochs was mostly just tuning what with a linear is a long polynomial to reduce the coefficients for all terms except the second.

This was the most simplistic model of them all, so moving on to a more complex dataset, this one where only a combination of values was allowed.

In this example, both values in positions 1 and 2 must be present, otherwise the model outputs false. Test data is as follows.

```
        A B C D E F G H I
TRUE  - 1 1 0 1 1 0 1 0 1
```

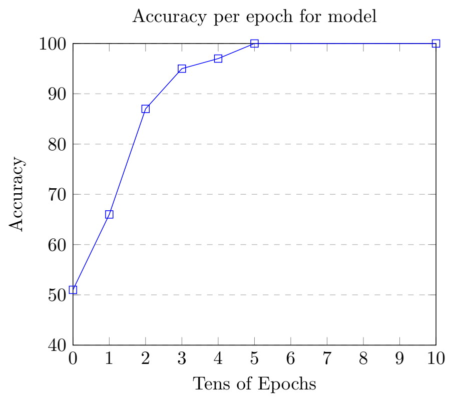

Figure 2: Accuracy versus Epoch, Initial Testing

```
TRUE  - 1 1 1 0 0 1 1 0 0
TRUE  - 1 1 1 0 1 0 0 1 0
TRUE  - 1 1 0 1 0 0 1 1 1
TRUE  - 1 1 0 1 0 1 1 0 1
TRUE  - 1 1 0 1 1 0 0 1 0
FALSE - 1 0 0 0 0 1 1 0 0
FALSE - 0 0 1 0 1 0 1 0 1
FALSE - 0 1 0 1 0 1 0 1 1
FALSE - 1 0 1 0 1 1 0 0 0
FALSE - 0 0 1 1 1 0 1 1 0
FALSE - 0 1 0 1 0 1 1 1 1
```

A dual layered model was used this time, instead of a single layered one in order too facilitate the training of multiple features.

```
def __init__(self, n_inputs):
    self.FL1 = nn.Linear(n_inputs, int(n_inputs/2))
    self.FL2 = nn.Linear(int(n_inputs/2), 1)
    self.activation = nn.Sigmoid()

def forward(self, x):
    x = self.activation(self.Fl1(x))
    return self.activation(self.Fl2(x))
```

This model took a similar amount of epochs to train, although instead of a massive increase in accuracy to begin with, it took a more traditional route of curving like a logarithm, rather than a super sharp increase at first with the polynomial flattening out later on.

While this is ideal test data, it is not very representative of the real world, where many many variations at a time can cause a disease to happen. While testing the rest of the test samples, from triplets, to singles plus doubles, meaning that one single gene could cause a disease, or a combination of two, as well as longer sequences,to try to emulate a full sequence, especially with far spaced out correlating genes, a DNN was not possible (feed forward style model, similar to the ones used in the past two examples) as it would mean no recursive calculation was allowed to be completed in the model itself.

These next examples include complex datatypes, ones that would be hard for humans to interpret, with either combinations of 2-3 genes, the presence of a singular gene and combinations of these factors.

For each of these, to simulate the actual frequency of genes, instead of using a 50% chance for each gene to show up, the frequency was changed to 0.1-5% per gene. With this adjustment, the length of the sequence was also changed too 4096. In terms of a delta from actual test data, genome frequencies are much accurate, however length

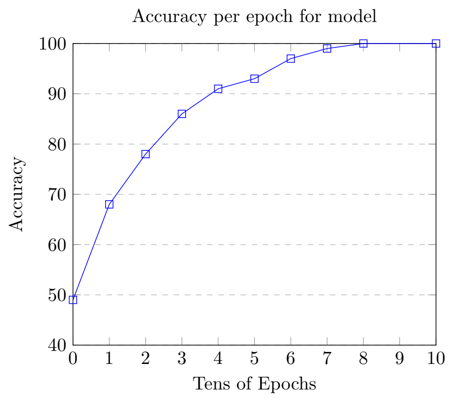

Figure 3: Accuracy vs Epoch second test

is still somewhat underrepresented. Before where an input string could be [0,1,1,0,0,1,0,0,1,1,1], the frequencies were adjusted so genes show up much less, more akin to a real genome, closer to [0,0,0,0,1,0,0,0,1,0]. Length was increased, as well as the amount of test sets.

Cross fold variation is a strategy to prevent a model from just memorizing train data and performing well on it, while doing a bad job with any data that is not in the test set. First it splits up the data into some number of batches, called K. The way it works is through repeatedly training the dataset while holding out a random batch, while training different models, making essentially multiple models all with their own bias. Then it combines them, so none can brute force all the training data. A training dataset of size 4096 was used, split into 4 cross fold variations of size 1024. Training was done iteratively, and test sets were evaluated each batch, batches were of size 32. The test set was of size 512. The summarized model is shown after k=4 cross fold variation.

2.1.1.1 RNN/LSTM GRAPHS

RNN layers reference past layers in a short term way. For instance, in most networks, going from node A to B to C to D, and then to an output. In an RNN the node C might check the value of A as well, hence the "recurring" part.

RNN's were experimented with one to four layers

All of these used the same generic template, but with less RNN layers and linear layers grabbing the last value.

```
def __init__(self, n_inputs):
    self.RNN1 = nn.RNN(n_inputs, int(n_inputs/2))
    self.RNN2 = nn.RNN(int(n_inputs/2), int(n_inputs/2))
    self.RNN3 = nn.RNN(int(n_inputs/2), int(n_inputs/4))
    self.RNN4 = nn.RNN(int(n_inputs/4), int(n_inputs/8))
    self.FL5 = nn.Linear(int(n_inputs/8),
                         int(n_inputs/16))
    self.FL6 = nn.Linear(int(n_inputs/8), 1)
    self.activationSig = nn.Sigmoid()
    self.activationSoft= nn.Softmax()
```

2.1.1.2 CNN GRAPHS

CNN graphs also reference older nodes, however do them in batches, to test different parts of the input data. In terms of testing, these were done almost exactly the same as the RNN's, with one to four layers, and used almost exactly similar code as above.

In General CNN graphs were a bit more accurate, and a better fit for the data, as well as more efficient. This would scale later over all the data.

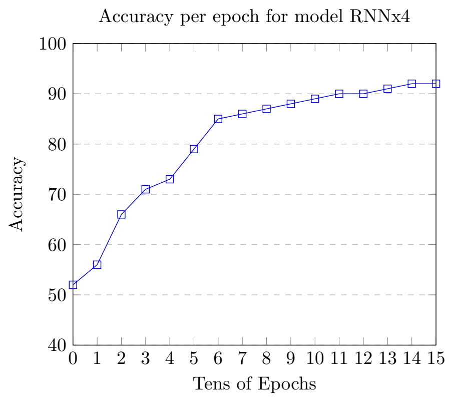

Figure 4: RNN Testing

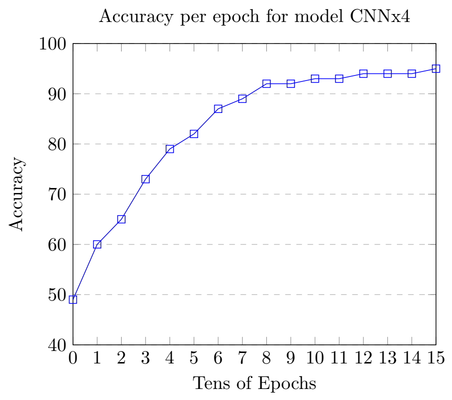

Figure 5: CNN Testing

2.1.1.3 Combination Graphs

The best out of the RNN+CNN Graphs was a single RNN layer three CNN layers and two MLP layers afterwards.

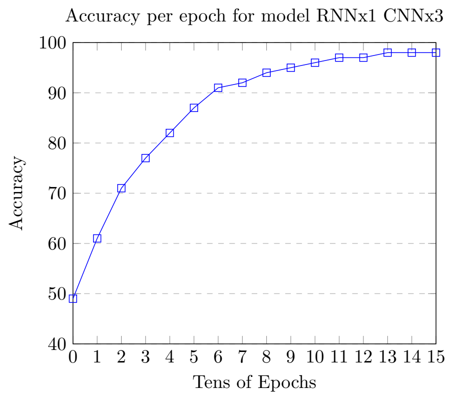

Figure 6: MLP, RNN, CNN Testing

As can be seen from all the more complex datatypes, learning rates are are much longer, with more epochs and larger batch sizes needed. Using a hybrid MLP+CNN model proved to have the most efficiency and accuracy in the long run, so a model with similar structure was decided to be used for the full analysis of the genome itself. Conclusions from artificial tests show that in general, deeper models perform better with CNN models able to singularly be somewhat better at predicting than RNN models, although a combination proved to be the most efficient model hitting above 95% accuracy in the quickest timeframe. Experimenting with the later layers proved to be pretty unnecessary, as 2 MLP layers worked well, with additional not adding any significant improvement.

2.2 Model Training and Evaluation Adenocarcinomas

2.2.1 Targeted Phenotypes

By using K=10 Cross Fold Variation, even with "smaller" datasets of around 1,000 to 4,000 samples, large amounts of testing and training can be achieved. The target was Adenoma and Adenocarcinomas, due to having the most data available by an order of almost 5 times to the rest of the diseases. 4,653 Cases were used, test data was aggregated among different cancers and healthy patients, as long as they did not have the targeted phenotype.

2.2.2 Data Gathering

Data was gathered from the TCGA FTP server, and categorized by dbSNP ascension number, firstly a dataset was compiled which went through all available files, categorizing SNPs. Most of this data was then converted to CSV format. In total 2,645,243 individual variations were found, and for variations that accounted for more than 1% of the population, 75,203 were found. Per person, around 40,000-50,000 individual variations were found. Values under 1% were eliminated to save computation time, the rest were again alphabetically characterized by dbSNP number and assembled into a dataset.

2.2.3 Model Training

For a pure classifier, this data would probably be fine, but in order to train a complex model to recognize the *correct* variations, shifting the files for the cases was required. After testing the model to high accuracy, it was then randomized.

The entire set was broken into k=21 cross fold variations, with 453 cases reserved for testing afterwards. Each of these cross folds had 200 cases each, allowing iterative testing between 21 different datasets. These sets were each iteratively trained on K folds, and their set to reserved accuracy was taken into account, with the summarized model taken to full accuracy. The model itself was run on Google Colaboratory, over GPUs. The models were run for 100 epochs at a time for each cross fold, before saving models locally to reset the environment or use a different account to regain access to GPUs.

The model itself was tweaked further, the basic Feed Forward network:

```
def __init__(self, n_inputs):
    self.FL1 = nn.CNN(n_inputs, int(n_inputs/2))
    self.FL2 = nn.CNN(int(n_inputs/2),
                      int(n_inputs/2))
    self.FL3 = nn.CNN(int(n_inputs/2),
                          int(n_inputs/4))
    self.FL4 = nn.RNN(int(n_inputs/4),
                      int(n_inputs/4))
    self.FL5 = nn.RNN(int(n_inputs/4),
                      int(n_inputs/8))
    self.FL6 = nn.Linear(int(n_inputs/8),
                      int(n_inputs/16))
    self.FL7 = nn.Linear(int(n_inputs/16), 1)
    self.activationSig = nn.Sigmoid()
    self.activationRel = nn.ReLU()
    self.activationSoft = nn.Softmax()
```

First as a sort of test the model was test against itself, the model was run without any cross fold variations, and while gained a high amount of accuracy against train data, suffered with test data.

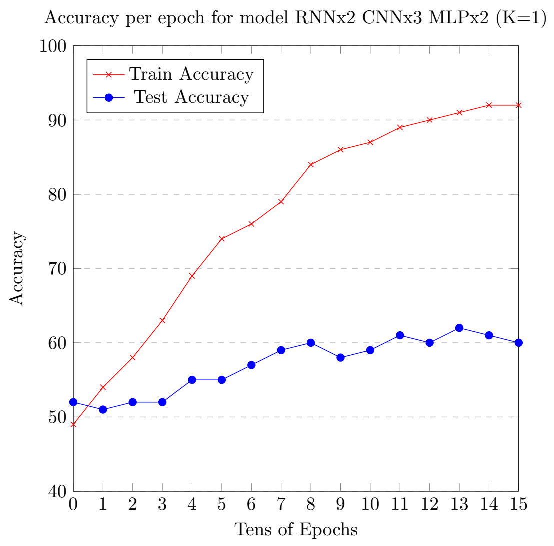

Figure 7: Unoptimized Graph (Overfitting Issue)

2.2.3.1 Cross Fold Testing

Evidently, even the highly randomized data from the final testing from CNN and RNN testing was not a match for the full set. Of course this set has k=1 cross fold variations, or essentially none, and with high levels of epochs it seemed to just overfit the train data. Randomizing trials met with most of the same result with test fluctuating between 55-63, close to random, with small amounts of insights. This makes more sense as accuracy started to stagnate as the model approached 80% accuracy, and probably hit brute forcing at around this time. With k=10 cross folds, the overfitting problem was somewhat reduced, allowing for a much nicer graph, although the model still wasn't completely optimized.

The test accuracy using cross variations was much better, definitely an improvement, but was still underperformed. Data-shuffling under the model proved to be somewhat useful, and by randomizing inputs for 120 separate tests the top 10% were significantly better, and allowing these to train each fold further yielded these results for the algorithm.

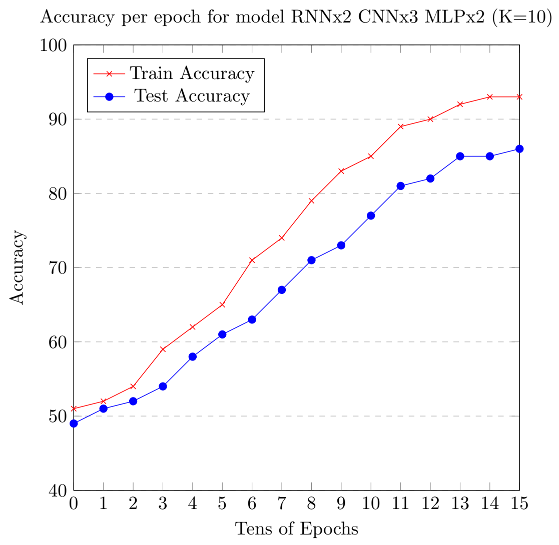

Figure 8: Cross Fold Graph

2.2.3.2 Layer Modifications

The model itself was tweaked, changing the number of nodes per layer and adding an additional linear layer at the end, so recursive layers could find patterns, and these patterns could be linked more easily in theory, and testing showed a small but relevant increase. Scikit-Learn and Scipy were both used to implement cross folds. Data was rescaled as well and from the source code it can be seen that both Sigmoid, Softmax, and ReLU were in use at times, with -1 - 1 normalized values found to work best. As for statistical analysis of the data itself, plotting top variations, most of them are around 2%.

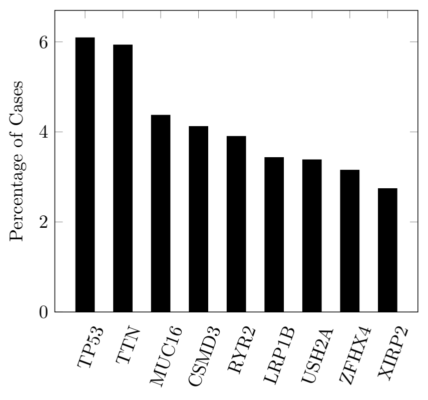

Figure 9: Frequency of Genes for Adenomas and Adenocarcinomas genomes

This means that most likely adjusting so only variations above 1% calculated while gives good computation time, accuracy could probably jump when adjusted to 0.1%. Computation time however also increased by a significant amount, and training models from 0.1% to 1%, and plotting results.

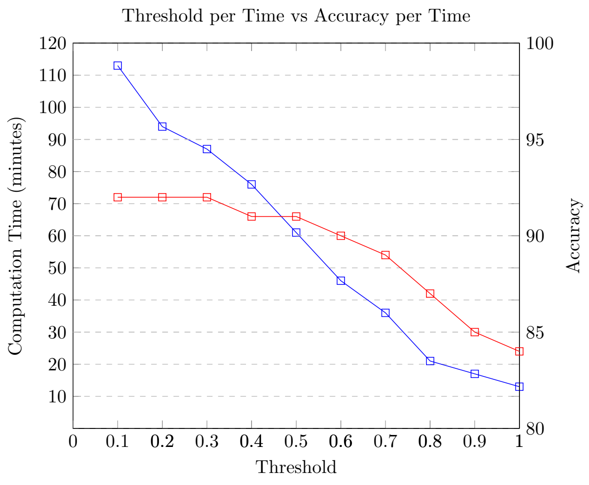

Figure 10: Accuracy versus Computation Time: Red for Accuracy, Blue for Computation Time

2.2.3.3 Hyperparameters

While 0.1% and 0.2% thresholds, allowing for almost any variation have much much larger n_inputs and total data sizes, they see no meaningful accuracy improvement, their extra layers probably only adding to the slight bump. Thresholds mean the frequency a gene must occur across the *entire* dataset in order to be part of the catalog, the percentage being the threshold. This percentage is usually very small. Threshold wise, training on a 0.7%, 0.5%, and 0.3%, for longer periods of time, with shuffled data, it was clear that 0.5 with a model with additional layers was optimal for training the most robust model which did best on test data. Continuing to train this model with long epochs and a high K value, gave probably the best results yet. Learning rate had a similar graph, with diminishing returns after a certain point, and 0.06 was the chosen maximum rate.

Weight Randomization:

Each machine learning model starts out with random weights as the coefficients for each node, and these are later tuned. However starting with different random weights can sometimes impact performance and model training times, so randomizing weights multiple times can help. For each cross fold variation, the initial weights were randomized 100 times, and after around 10 epochs of testing, the best 5 were taken from each fold and trained to completion, the highest test accuracy one the final model.

Input Randomization:

Input order was randomized over 1000 trials, for them after 3 epochs for each candidate. the top 100 were continued to 10 epochs, and after that the top 10 were continued to completion, the top five of these taken for use with backtracing later. The top models training is shown in all graphs.

Additional Steps:

Ensemble tuning and multimodel predictions may have seen an improvement, although backtracing to figure out what the models were seeing would be further complicated by this to the point where it was decided not to be used. In the end a couple of CNN models to aggregate the sequence itself, followed by a couple of RNN's to compare the CNN data, as cited in literature was the best model structure, in addition to testing throughout this study.

Since after this training run, this was the final "top performing model", model results can be seen in the results section.

2.3 Backtracing and Model Reconstruction

In the world of Machine Learning, there are a couple of known facts, the first being that models train in a sort of black box style, meaning that there is no real way for researchers to know or understand what is going on inside the model or how it makes its predictions. For all we know, it could be completely randomly guessing and getting answers correct for no reason at all. Most projects involve finding a dataset which was only statistically analyzed before using a basic machine learning model to increase classification performance by a certain amount, and then presenting findings. This may be useful in felids such as diagnostics based on symptoms or doing work formerly done by humans such as image recognition, however it is limited to quantifiable areas like these.

In more qualitative areas where just a classifier would not help the science advance, in actual research, machine learning models are not helpful. They do not help us understand why certain things are happening, and can only point out that things are happening. Classifiers are good for automation, but not for research, at least in felids such as motif and variant discovery, being one of the key reasons they are not widely used these days.

Determining how an algorithm works however is extremely complicated, requires serious amounts of either computing power or time. Both of which are the point of machine learning models. Avoiding large amounts of computation time is one of the reasons machine learning models are useful to actually use to research data, and find correlations, as it does not require long statistical checks. Smart backtracing algorithms are necessary. In this next subsection, I will experiment with two to three different algorithms with both proof of concept data and actual data to determine the best one to use.

2.3.1 Differnet Model Reconstruction Algorithms

2.3.1.1 Input Manipulation

The first type of reconstruction algorithm is input manipulation, an algorithm in which the inputs are literally manipulated to see which inputs have a serious impact on the model. For instance, randomizing 0's and 1's for the input can easily expose which outputs are chosen with highest confidence, and show which inputs create those outputs. The only issue with this type is that it has a very high computation rate. This is because for instance with our model of after normalization to 0.5%, around 73,000 binary inputs, manipulating these would take $2^{73,000}$, which is far far far over a time that would be manageable in a timeframe of years, even with a supercomputer.

This traditional option is definitely not feasible. However, as a test run, for all of those genes that highly correlate throughout different studies of the disease I decided to manipulate data to trigger these genes to see if the model was able to successfully correlate with certain genes.

Below are tabulated results for top genes. These results were tabulated by randomizing all other variables, keeping these genes selected, over 100 trials, and averaging the final output (normalized between 0 and 1, with 0 = not present, and 1 = present)

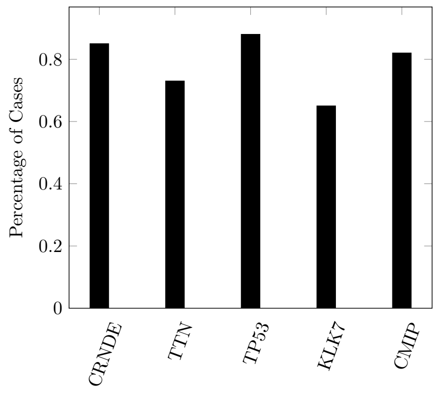

Figure 11: Model Predicting Adenomas Highlighted Genes

The final results are promising, showing that the model agrees with conventional wisdom, and selecting certain genes that are known to correlate produces positive results. Selecting random genes, or the wrong ones has little to no effect, keeping results normalized around 0-0.3%. By having 100 trials random gains which could happen from unknowingly triggering a correlative gene how unlikely, was mitigated.

2.3.1.2 Weight Tracing

Instead of brute forcing multiple different models, the best way to approach deconstructing ML models, is through looking at weights, as well as tracing weights close to or at 0 meaning we can eliminate entire portions of our model. Weight Tracing was done by starting at the final level, and working backwards to assign a weight to each node pathway, based on the nodes before it.

For instance, see this node graph.

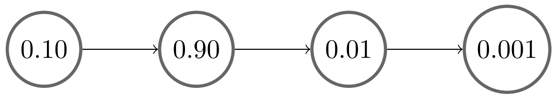

Figure 12: Simple Backtracing Figure

In this very simple graph, tracing from left to right, we can disregard the input, as by the end of the sequence the nodes value has been minimized, almost completely.Similarly we can section off multiple parts of the model by tracing back any super small coefficient numbers, and assigning each path a coefficient for the total importance across the entire model, and eliminating any under a certain threshold. This can be done programmatically, parsing through the model converted to an array, while eliminating any odd pieces.

In this model for instance, making D_4 a weight close to 0, eliminates all nodes that do not feed into E_4, essentially in this case just A_1, B_2, C_3. Say C_2 is also close to 0. Then B_2 would also be eliminated. Similarly, one could multiply out coefficients an all of the rest to find the full weightings for all values. For instance, in the simple feed forward network seen above, for input X, it passes through 4 simple weights for an output of 0.00000009. Of course, with connected networks, the output or output up to a point, would factor in every other variable, although with coefficients and the fact that variables are in a binary fashion, makes adding coefficients easier.

The final strategy incorporates some input manipulation, and some "reaching in" to the model to see what variables are changing. The weights of each node is recorded, and changing each of the 14,223 nodes, now in a much smaller ML network, will have an effect on the model itself. For each input, as it is toggled, each node value is recorded changing with either all other inputs triggered, or all other inputs not triggered. Through this method for each node checking which nodes it affects, even if it does not hit a threshold for those nodes, and later simplifying, makes the node process simplified.

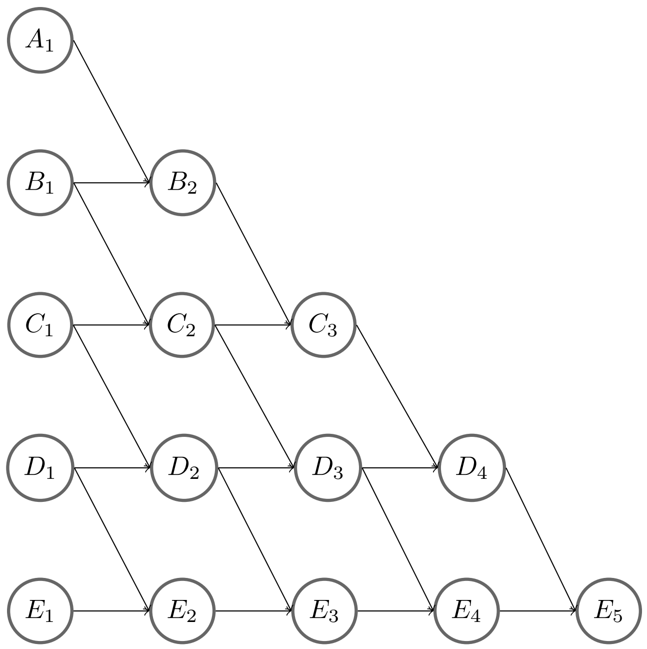

Figure 13: Complex Backtracing Explanation Figure

This technique is close to the one used by Simonyan et. al. in their Saliency maps, generating the data for each "pixel" in this case black or white binary dot, due to their weights, and determining relative importance, almost exactly as the method I used. This technique was appended by Montavon et. al., adding heatmaps per each layer based on importance towards the future layers.

[Simonyan K, Vedaldi A, Zisserman A. Deep inside convolutional networks: visualizing image classification models and saliency maps. The 2nd International Conference on Learning Representations, Banff, AB, Canada, 2014.]

Assigning these importance levels in sort of a backwards fashion can help show which nodes are most important, as well as trace all the way back into the input nodes.

2.3.2 Applying to Model

The model was first worked with a sort of heatmaps style algorithm as outlined above, code which took in the models weights, and then assigned an importance at each level. This importance was also coupled with an additional number, taken from testing the model over 100 randomized trials for each node, so layer by layer toggling nodes for randomized other nodes to see the impact the node had. This was done in a backpropagation method too. With these two numbers, a theoretical backpropagation as well as a practice experimental backpropagation, weights below a certain threshold, or that were not connected to a major node, were disabled. Thorough the model's total of about 250,000 nodes in all layers, around 83% were removed, leaving only around 32,500 nodes total. Disabling these nodes required a bit of training to update the model to realize that all these other nodes were gone, although training was done iteratively, first removing nodes under a small threshold, before increasing the threshold to its final value to allow the model to slowly adapt.

Finally, with only about 30,000 nodes left, still with an input of around 70,000, any input which had no correlations with the model after removing intermediary nodes was eliminated as well, leaving only around 12,224 inputs. Using transposed convolutional layers to analyze the remaining part of the model, Conv2DTranspose was used in this case, with outputs from the convolutional layer transposed to a 2d object for Conv2D to work. Finally, all of these nodes were retrained on a smaller model, using no weights from the last time, and the heatmap algorithm was applied again, to rank the new nodes in order of importance.

This entire process, due to anomalies in testing, was done for the top 5 models, after the randomization section in the above section about model training, the top 5 performing models were kept, and this entire process was performed on all of them. The averaged value without outliers, is whats pretended at the end.

3 Results

3.1 Model Results

In the end, the model had inputs of 74,553, a threshold of 0.5%, over a test set of 4,653 split into 10 cross fold variations. Among the remaining cancers was picked another random 4,653 cases, for data of the phenotype not present. For each holdout, data position was randomized for each of 40 epochs until maximum training was achieved, and the model was repeatedly saved to assist in early stopping to prevent overfitting. Three Convolutional Neural Network Layers wer used, followed by Two Recurrent Neural networks, followed by 3 MLP or Feed Foreword layers. Input weights were randomized to see which ones would provide the quickest learning for each fold.

For each dataset: 75,203 variations were identified to be common, and input was a tensor of size 75,203 of binary numbers representing either a variation present or not. Final accuracy for the model was 93.4% after 257 epochs, on Test Data. After the first heatmap pass, the model was reduced to 12,224 input layers, the training graph for the reduced graph and model was trained the same way, and compared with the original.

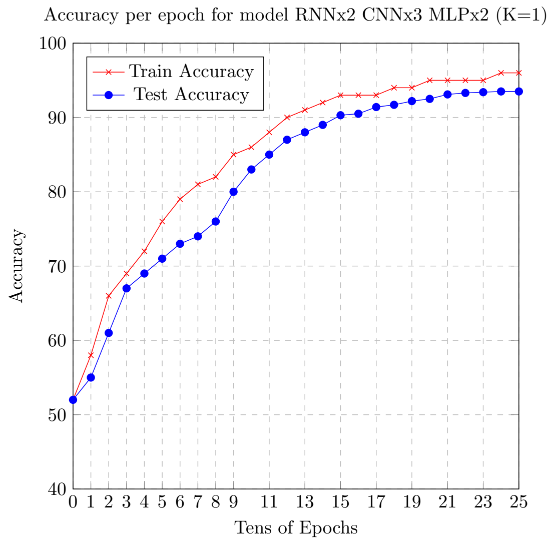

Figure 14: Final Training Graph

The model was trained over 4 accounts on Google Colaboratory, total training time for this model was 19 hours spread across two GPU sessions.

In total, working on multiple projects at once, total training time used and resources were about 374 hours, over a span of two weeks.

3.2 Genes Identified

Following methods above, the following genes were identified over the model algorithm. The results from the first heatmap pass, written vertically are shown, almost all lighter areas were eliminated, for a size of 12,224. (This is the heatmap for the inputs, layer based heatmaps are too large and numerous to display, while not providing much useful data)

Heatmap for second pass, this time un-averaged, showing much less detail, as there are less inputs.

Averaging the input weights and ordering them, manual analysis was performed on all top 1,000 markers, both for the input layer, and for the end of the first convolutional layer. Manual analysis includes going through each identified gene, and looking at its statistical position, as well as other genes it is present with in the convolutional layer, to identify if it must be paired later down the line in the model, or is just random. More data and testsets as well as training over the same epochs many times with an unbalanced dataset helps make these more definitive. Final results are summarized below. Occurrence position is also taken into account.

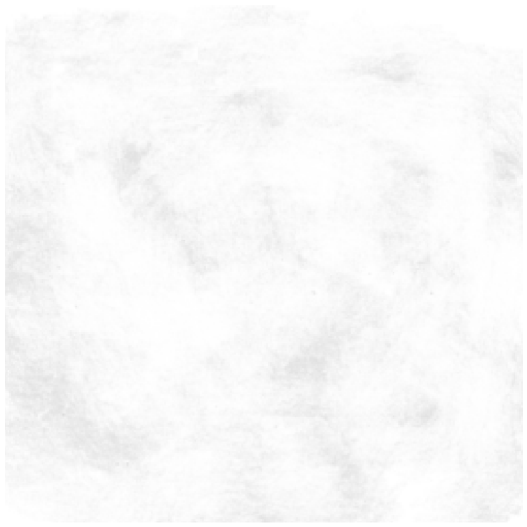

Figure 15: First Heatmap Importance Identifier

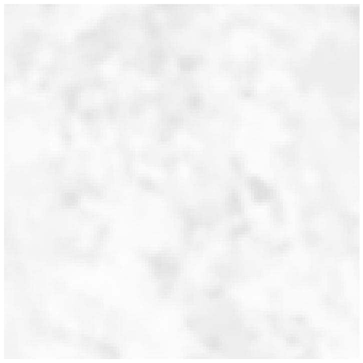

Figure 16: Second Heatmap Importance Identifier

Gene	**Importance**
TTN	0.941
TP53	0.934
MUC16	0.913
RYR2	0.892
COL12A1	0.846
CSMD3	0.845
DNAH7	0.832
APC	0.831
CUBN	0.829

Now these genes are nothing new, they are already known to have a high amount of correlation, but of the genes without a high correlation which had high importance. Of each of these, all of them were also looked at manually, and made sure to have at least somewhat high amounts of accuracy when they appeared. There are many more, although only a couple were summarized, the rest are awaiting manual analyses before summarizing. In total the first 30 were taken. In terms of pairing these together, to find which combinations work, manual analysis is still in progress, but I am sure some of these super high importance genes have to be placed with a different one for maximum effect.

Gene	Importance
TPRM16	0.762
TCHH	0.632
ST18	0.621
SMARCA3	0.610
POTEG	0.598
BOD1L1	0.597
DSCAM	0.573
DIDO1	0.567
CDH9	0.525
PAK7	0.492
ABCC1	0.484
TPO	0.482
CUX2	0.481
MED12	0.481
CHD7	0.480
BSN	0.479
SCN1A	0.479
OR2T4	0.478
NBEA	0.477
MTUS2	0.476
BRINP2	0.475
DOCK10	0.473
DSG3	0.471

4 Discussion and Conclusions

The task of detecting variations is difficult, requiring either only looking for certain markers, in the traditional statistical approach, or to do pairwise checks, immense computation power is required. As can be seen from this paper, after determining and creating a model, and then using backtracing, it was possible to determine many new variations. Automating the process has helped identify many new correlative genes, and spending a detailed amount of time tracing each of these identified genes, it is possible to manually identify 20-30 “sections” of genes that correlate with the phenotype. Most machine learning models have no way of presenting their findings, and as had to be seen, one had to be trained and retrained many times before getting a model which actually had predictive and analytical power, and able to execute on testcases. Testing the model on known genes showcased that it had found these correlations itself, and was able to do so very efficiently and quickly. In the end, the idea behind the study was simple, train a model, and then try to use methods to figure out what that model trained, but the results were better than expected, as is with most machine learning.

In this study I used a heavily tuned machine learning model trained on a dataset that had a high accuracy on test datasets, the model itself was tested on Adenomas and Adenocarcinomas, and in addition to successfully identify a large amount of genes that already were known to have correlation with prevalence of the disease, some new variations which were not previously known to correlate. Manual analysis of these showed that they were most likely at least somewhat correlative. For most of these, with some manual analyses most of them showed up in test sets and after combing through every time they showed up, it was simple to identify that all of these worked.

Additionally, through the process of training the model, I was able to determine exactly what type of model works best in these applications, and the best way to implement it, as well as data preparation. As summarized throughout Model Training and Backtracing, create the foundations and programs to run this software is the majority of the work is complete. Through the processes of backtracing, which while took time and lots of computation, I was able to finally use machine learnings to determine results, instead of just classifying or taking on empirical tasks.

I think the main limitation of this study is just stretching the amount of data. While with the strong correlations drawn from the model, for the 70,000 or so parameters, very few were in use per model, and having a similar cutoff, 0.5% for a much larger test set, possibly 10,000 to 20,000 variations would produce much more results. Testing on the other datasets, which had even less, showed that the model was more off, with many more duds implemented into its testing, and a larger overfitting issue, even with cross folds and setting aside certain data. In the end however, the results were admirable, and being able to count on one hand the amount of prior studies into using models to identify variations through similar methods to mine, although of course each taking a different route and none focusing on genetic variations, it is great to start work in a new way.

I would like to continue applying this model to larger test sets as well as diving deeper into the variations identified, for most of them, simple analysis was needed to know whether or not that variation was useful, although even more analysis, as well as using other models from separate to test the significance on datasets independently would be useful. In addition, with ML models able to take any input, if I could combine variation data with more data types such as methylations biospecimens and transcriptome profiles, as well as other datatypes, to create more comprehensive studies, and combine these datatypes in studies that have never been done before.

I would say above all, even above the new variations identified from this study, the main conclusion drawn can be the efficacy and prevalence of Machine Learning models to tackle all sorts of biological data, even less empirical more qualitative felids such as Bioinformatics, which require a large amount of research and are much more results based than most applications of Machine Learning.

5 References

Al-Ajlan, Amani. 2018. "CNN-MGP: Convolutional Neural Networks for Metagenomics Gene Prediction." Interdisciplinary Sciences: Computational Life Sciences. December 27. https://link.springer.com/article/10.1007/s12539-018-0313-4.

Dias, Raquel. 2019. "Artificial Intelligence in Clinical and Genomic Diagnostics." Genome Medicine. November 19. https://genomemedicine.biomedcentral.com/articles/10.1186/s13073-019-0689-8.

Eraslan, Gökcen. 2019. "Deep Learning: New Computational Modelling Techniques for Genomics." Nature Reviews Genetics. April 10. https://www.nature.com/articles/s41576-019-0122-6.

Hashim, Fatma. 2019. "Review of Different Sequence Motif Finding Algorithms." PubMed Central (PMC). June 1. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6490410/.

Jackson, Maria. 2018. "The Genetic Basis of Disease - PubMed." PubMed. January 1. https://pubmed.ncbi.nlm.nih.gov/30509934/.

Johnson, Andrew. 2009. "SNP Bioinformatics: A Comprehensive Review of Resources." PubMed Central (PMC). October 1. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2789466/.

Kopp, Wolfgang. 2020. "Deep Learning for Genomics Using Janggu." Nature Communications. July 13. https://www.nature.com/articles/s41467-020-17155-y.

Libbrecht, Maxwell. 2015. "Machine Learning Applications in Genetics and Genomics." Nature Reviews Genetics. May 7. https://www.nature.com/articles/nrg3920.

Liu, Jianxiao. 2020. "Application of Deep Learning in Genomics." Science China Life Sciences. October 10. https://link.springer.com/article/10.1007/s11427-020-1804-5.

Marees, Andries. 2021. "A Tutorial on Conducting Genome-Wide Association Studies: Quality Control and Statistical Analysis - PubMed." PubMed. February 27. https://pubmed.ncbi.nlm.nih.gov/29484742/.

panelLefterisKoumakis, Author. 2021. "Deep Learning Models in Genomics; Are We There Yet?" Accessed January 29. https://www.sciencedirect.com/science/article/pii/S2001037020303068.

Paszke, Adam. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." ArXiv.Org. December 3. https://arxiv.org/abs/1912.01703v1.

"Patient-Directed Insights on the Evolution of Treating Rare Genetic Diseases." 2021. Accessed January 29. https://www.longdom.org/proceedings/patientdirected-insights-on-the-evolution-of-treating-rare-genetic-diseases-20220.html.

Przystalski, Karol. 2020. "Machine Learning in Genomics. Artificial Intelligence — Codete Blog." Codete Blog - We Share Knowledge for IT Professionals. March 26. https://codete.com/blog/machine-learning-genomics/.

"PyTorch Documentation — PyTorch 1.7.0 Documentation." 2021. Accessed January 29. https://pytorch.org/docs/stable/index.html.

Talukder, Amlan. 2020a. "Interpretation of Deep Learning in Genomics and Epigenomics." OUP Academic. August 20. https://academic.oup.com/bib/advance-article/doi/10.1093/bib/bbaa177/5894987.

———. 2020b. “Interpretation of Deep Learning in Genomics and Epigenomics.” OUP Academic. August 20. https://academic.oup.com/bib/advance-article/doi/10.1093/bib/bbaa177/5894987.

Webb, Sarah. 2018. “Deep Learning for Biology.” Nature. February 20. https://www.nature.com/articles/d41586-018-02174-z.

2021. Accessed January 29. https://www.longdom.org/open-access/a-critical-survey-of-mathematical-approaches-towards-genome-and-protein-sequence-comparison.pdf.