BMI 203 Final Project

Matt Jones

March 23, 2018

Contents

1 Introduction **1**

2 Autoencoder **1**
2.1 Methodology . . . 2
2.2 Results . . . 2

3 Classifying Transcription Factor Binding Sequences **3**
3.1 Representing Genomic Sequences . . . 3
3.2 Filtering the Negative Data . . . 4
3.3 Architecture of the Neural Net . . . 4
3.4 Evaluating the Effect of Training Data Bias . . . 4
3.5 Training Regime . . . 5
3.6 Results . . . 6

4 Conclusions **7**

5 Code Availability **8**

1 Introduction

Transcription factor binding prediction is a particularly difficult task yet important for understanding the functional consequences of genetic variation in noncoding regions. In this assignment, I have adapted an artificial neural net framework for the task of classifying DNA sequences as sites for transcription binding or not. The data used to generate this classifier is from yeast, and in particular from regions 1KB upstream from genes.

I will first describe my artificial neural net framework and discuss its functionality in the case of 3 layer autoencoder. I will then analyze its performance on a training set of genomic sequences labeled as binding sites or not. I will discuss my techniques for verifying its performance as well as crucial design decisions. I will end by considering potential improvements to the model.

2 Autoencoder

In this section I will describe my $8 \times 3 \times 8$ autoencoder used to develop my artificial neural network. In particular, I developed a fully connected feed forward neural network with a single hidden layer consisting of 3 neurons, with standard sigmoidal activation functions. Both input and output layers consisted of 8 neurons, and both the input and hidden layers included a bias term to the next layer. This autoencoder was used to encode a length 8 binary vector into a 3 dimensional space; mathematically, we sought to perform the following:

$$\begin{aligned}
10000000 &\rightarrow h_{1,1}\ h_{1,2}\ h_{1,3} \\
01000000 &\rightarrow h_{2,1}\ h_{2,2}\ h_{2,3} \\
\ldots &\rightarrow \ldots \\
00000001 &\rightarrow h_{8,1}\ h_{8,2}\ h_{8,3}
\end{aligned}$$

2.1 Methodology

As mentioned earlier, I employed a fully connected 3 layer neural network to complete this task with sigmoidal activation functions for each neuron. Specifically, each neuron (besides the input layer) produced an output according to:

$$\sigma(y) = \frac{1}{1+e^{-y}}$$

where y is the input to the neuron. In the case of my neural net, y is defined as

$$y_i^{(l)} = \sum_j \beta_{j,i}^{(l-1)} x_j^{(l-1)}$$

where we have defined the input to the i^{th} component of the l^{th} layer. In the mathematical definition above, $\beta j, i^{(l-1)}$ is the weight connecting the j^{th} element from the previous layer to this i^{th} component we are concerned about, and $x_j^{(l-1)}$ is the output of the j^{th} element in the previous layer. The "feedforward" step consists of calculating all outputs for all neurons through simple matrix algebra. To note, we initialize by setting all β to some small random factor.

Upon passing our inputs through the network with the feedforward step, we then would like to conduct "backpropagate" to update weights according to the error produced at the end. In short, we'd like to calculate the amount of error that each weight contributed, and correct the weight proportional to this error. Although I will not derive the backpropagation algorithm and its updates, it stems from the query $\frac{\partial E}{\partial w_i}$ and conveniently for sigmoidal activation functions, the derivative is simply $\sigma(x)' = \sigma(x)(1-\sigma(x))$. This makes our updates very simple for our three layer network. Specifically, for each weight connected to an output node, we can find the portion of error it contributed , δ_k:

$$\delta_k = o_k(1-o_k)(t_k - o_k)$$

where o_k is the output of the k^{th} output node and t_k is the true label of this node. We can also find the portion of error that each hidden layer node contributed to the error, δ_h:

$$\delta_h = o_h(1-o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

where o_h is the output of the h^{th} hidden node and the sum represents the sum of downstream contributions it had to the outputs. Finally, we update weights accordingly:

$$w_{i,j} = w_{i,j} - \alpha \delta_j o_i$$

where α is a learning rate specified as a hyperparameter.

2.2 Results

Now we turn to the results of the autoencoder built using the methodology described in the previous section. As discussed above, the neural net will be tasked with representing the 8 dimensional input in some 3 dimensional latent space, which I can extract from the hidden layer. Before evaluating the values of the latent space corresponding to each of the inputs, I sought to find the optimal training parameters and thus analyzed the sum of squared errors as a function of both the learning rate and the number of epochs I allowed my neural net to train for. I conducted this analysis 5 times and report the mean squared error for each pairing of learning rate and number of epochs (Figure 1). From the data, it seems that the error rate of the model reduces dramatically with both the learning rate and the number of epochs, as one would expect. To minimize the amount of time needed to train my model, I chose to use a learning rate of $\alpha = 0.05$ and number of epochs $N = 100000$. In doing so, I have a perfect autoencoder and the hidden layer's representation of the latent space is the following:

$10000000 \rightarrow 0.943012\ 0.987183\ 0.00584752$

$01000000 \rightarrow 0.994185\ 0.986761\ 0.990468$

$00100000 \rightarrow 0.923276\ 0.00458667\ 0.979986$

$00010000 \rightarrow 0.0150614\ 0.971213\ 0.990424$

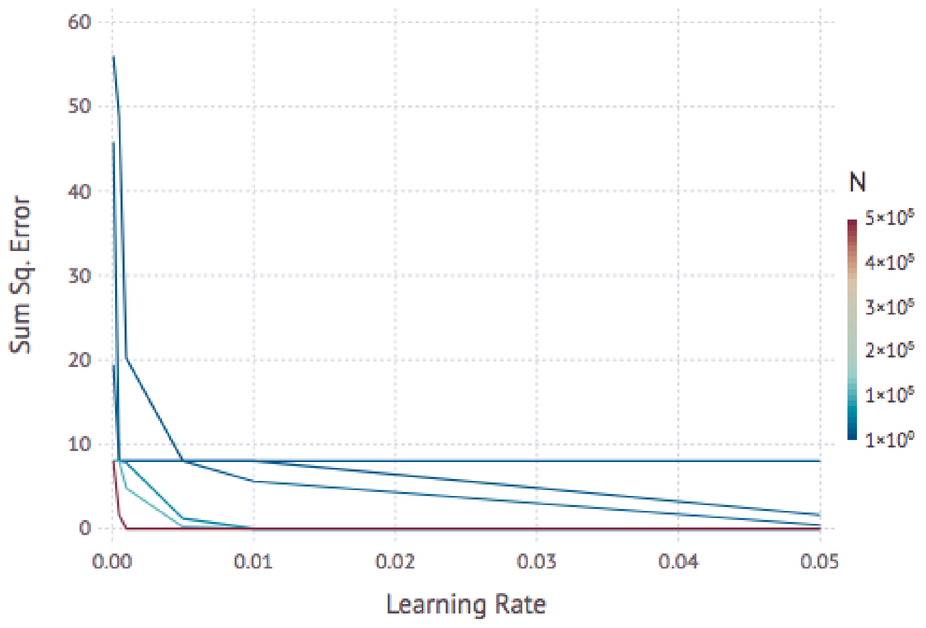
$00001000 \rightarrow 0.00729402\ 0.973194\ 0.0706169$

$00000100 \rightarrow 0.00676055\ 0.0171775\ 0.822166$

$00000010 \rightarrow 0.980623\ 0.0289271\ 0.0374664$

$00000001 \rightarrow 0.0387514\ 0.0190166\ 0.00535882$

Figure 1: $8 \times 3 \times 8$ Autoencoder Squared Error Determined by Hyperparameters



3 Classifying Transcription Factor Binding Sequences

After developing the autoencoder framework discussed above, I turned to creating an artificial neural network for classifying transcription factor binding sequences for yeast. Essentially, I sought to develop a framework that could give the probability that a sequence was a binding site for some transcription factor based on 17 base pairs. I will first discuss here the decisions I made regarding the representation of the DNA sequences and the architecture as a whole. I will then describe the training regime that I used to test my model, as well as the experiments I conducted to verify that my model was producing reasonable results.

3.1 Representing Genomic Sequences

As mentioned above, the neural net here is tasked with classifying sequences of DNA - specifically, a string of 17 letters from the alphabet $\mathcal{A} = \{A, C, T, G\}$. To represent the sequence, I elected to first convert each letter to a one-hot encoding as such:

$$\begin{aligned} A &\rightarrow [1,0,0,0] \\ C &\rightarrow [0,1,0,0] \\ G &\rightarrow [0,0,1,0] \\ T &\rightarrow [0,0,0,1] \end{aligned}$$

Thus, I create a mapping from each sequence to a 4×17 matrix encoding the sequence. To avoid having to deal with tensors in my model, I then flatten the 4×17 matrix into a one dimensional vector which is then passed to my neural net. For example, the sequence $ACTG$ would be converted to $[1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1]$. To note, information is not lost during this transformation: I can easily reverse the process and recover the original sequence by converting the one-hot encodings of each 4-bit chunk in the vector back to its respective nucleotide.

To note, the "negative" test data is provided in the form of a set of 1K base pair upstream sequences for yeast. To convert this into a form that can be used to compare against the "positive" data, I apply the following procedure: first, I create chunks of length 17 strings from the 1K sequences. I then convert these length 17 strings into the one-hot vector that I described above.

Labels are simply created by marking "1" for binding sites (i.e. for those sequences provided in the positive data set) and "0" for non-binding sites.

3.2 Filtering the Negative Data

It should be noted that the negative data is merely genomic sequence data from promoter sequences upstream from genes in yeast. As a consequence, the data is not necessarily unique nor negative (i.e. there may be subsequences in this data set that are actually binding sites). To filter the negative data to obtain truly non-binding sites I employ a simple hashing filter system.

Specifically, to avoid an involved look-up scheme where I must do some 17 base pair comparisons per pair of sequences, I create a simple hash function that I use to map each sequence uniquely and then look for conflicts. The hash function for some length n vector x I use is simply:

$$h(x) = \sum_{i=1}^{n} x_i 2^{i-1}$$

Because we are guaranteed that all values will be either 0 or 1, intuitively a good hash function is just the conversion of this bit string into its base-2 integer.

To then filter, I do a simple integer lookup for all values of my positive sequences in my negative sequence collection; finally, I remove duplicates from the negative sequences. Thus, I have created a coherent set of data to train my neural net.

3.3 Architecture of the Neural Net

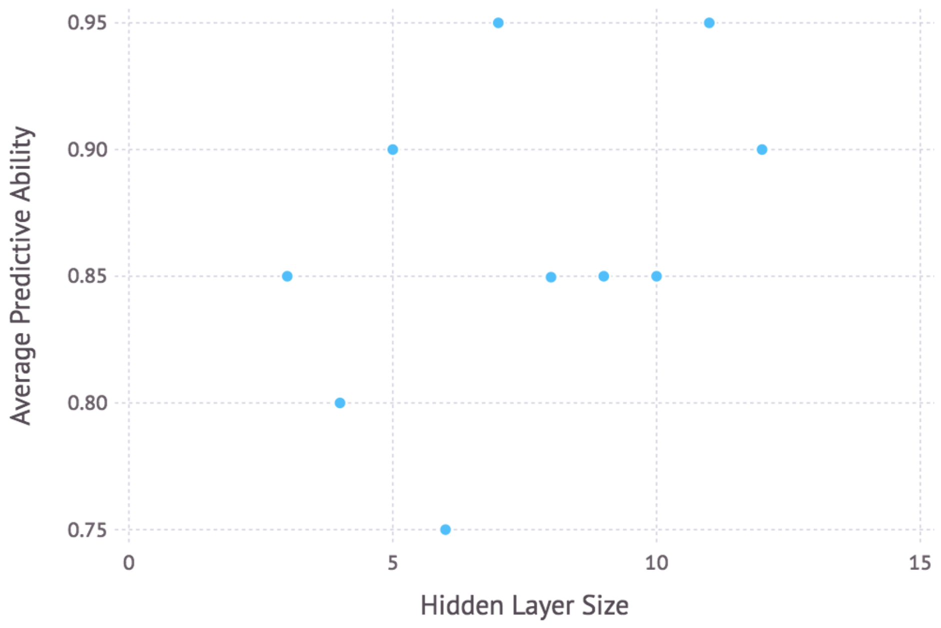
I used a three layer neural net to classify the sequence data, much like with the autoencoder. Notably, I only have a single output node, corresponding to the probability of the site being a binding site or not, and the number of neurons in the hidden layer is no longer 3. In fact, to decide on the number of neurons to use in my neural net, I conducted a simple analysis which analyzed the square error as a function of the number of neurons in the hidden layer for a consistent 1000 epochs with a 0.05 learning rate (Figure 2). To note, I ran this analysis on a balanced set of data, consisting of equal numbers of positive and negative sequences, to avoid any test bias issues.

There seems to be no correlation between hidden layer size and predictive performance. As such, I arbitrarily selected a hidden layer size of 11, one of the two parameterizations leading to an average predictive power of 95%. To conclude, I employed a three layer fully connected neural network with 168 (17×4) input nodes, 11 hidden nodes, and a single output node, all with sigmoidal activation functions.

3.4 Evaluating the Effect of Training Data Bias

I next set out to test the effect of training data bias on my classifier. Specifically, because there is much more neutral (i.e. non-binding) genomic sequence than binding sequence, I was curious to see

Figure 2: Predictive Ability as a Function of Hidden Layer Size



what the effects of biasing the training set by different amounts of negative sequence. To do this, I tested the ability of my neural net to learn features to discriminate between types of sequences over a range of biases. I did this by training the neural net over 1000 epochs with a learning rate of 0.05 for a range of biases, and reporting the final predictive ability. I *do* not use cross validation here because all I am testing is whether or not the bias in training data precludes the ability of my network to learn discriminatory features. I defined "bias" in this term as the factor corresponding to how much more negative sequence I had than positive sequence - for example, if I had p positive sequence, then a bias of 2 would correspond to $2p$ negative sequences. My results are presented in Figure 3.

As one would expect, the ability to learn is heavily impacted by the class inbalance problem discussed above. Surely, by augmenting the learning paramters may be able to overcome some of this class inbalance, but for the purposes of training the neural network I will focus on providing only balanced test sets, and shuffling which negative examples the neural network receives each iteration.

3.5 Training Regime

To test against overfitting, I employed a cross-validation scheme to analyze how well my neural net was learning the data. Specifically, each iteration of cross-validation, I selected 90% of the data to be training and left out the other 10% to be testing data; I then allowed my neural net to learn for some N epochs with a learning rate α and evaluated the performance on the left out data. Using this method, I can obtain a distribution of predictive ability for each set of learning parameters. In Figure 4, I report the mean predictive ability over 10 rounds of cross validation for a set of epochs and learning rates; to note, I use only use balanced training data to avoid confounding by class inbalance in finding the optimal learning parameters. After my analysis, it's apparent that using higher learning rates with a greater number of epochs is ideal for learning the sequence type - as such, I will use $N = 5000$ and $\alpha = 0.05$ from now on.

After finding the optimal set of parameters, I then turned to evaluating how sensitive these parameters were to class inbalance. As such, I conducted a very similar analysis to that in the previous section where I quantified the average predictive value over 10 cross validation rounds for various degrees of bias. My findings presented in Figure 5 demonstrate that, non-intuitively, the class inbalance problem does not heavily impact the ability of the neural net to classify sequences.

Figure 3: Predictive Ability as a Function of Class Inbalance

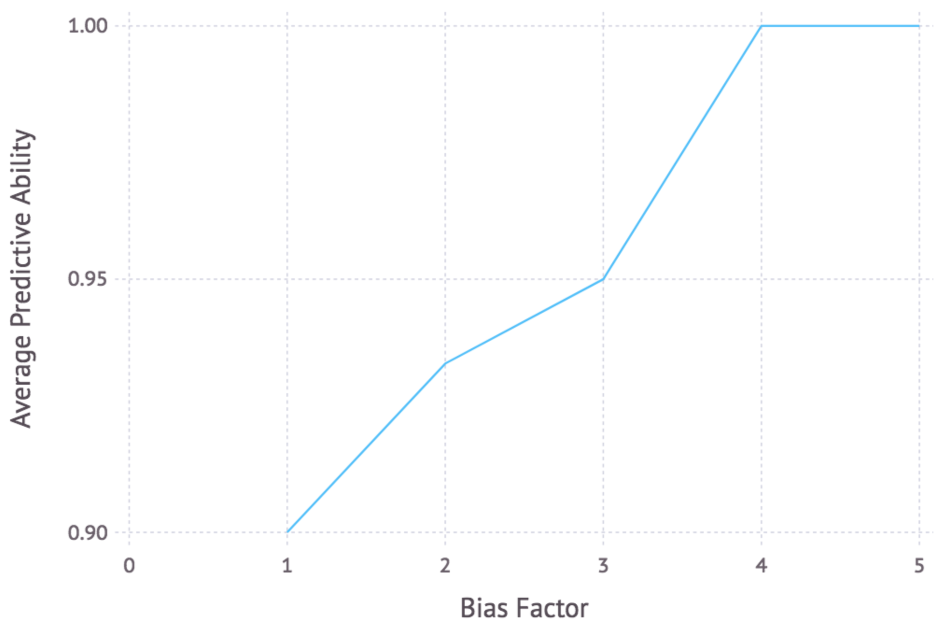
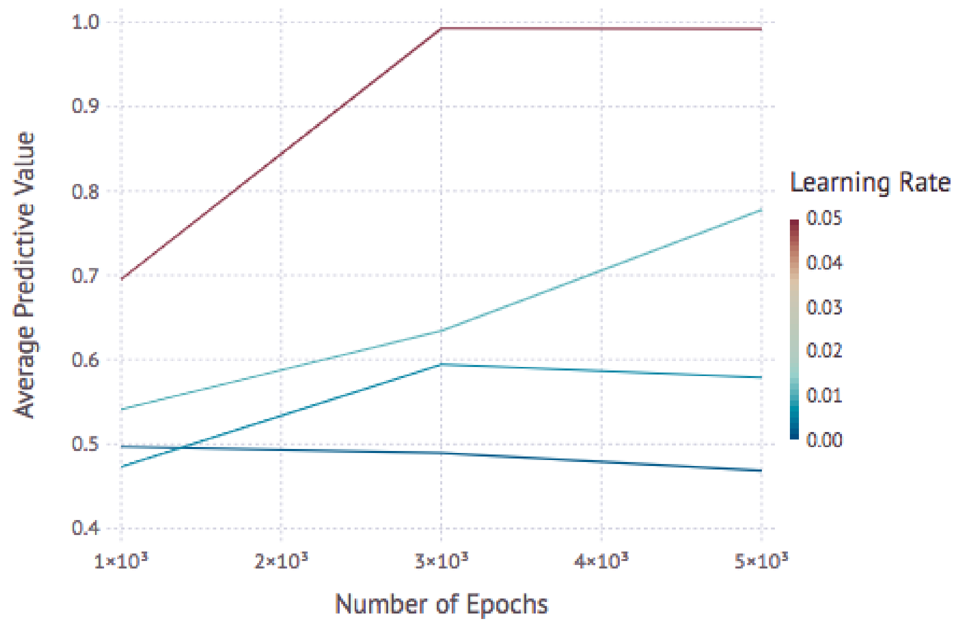


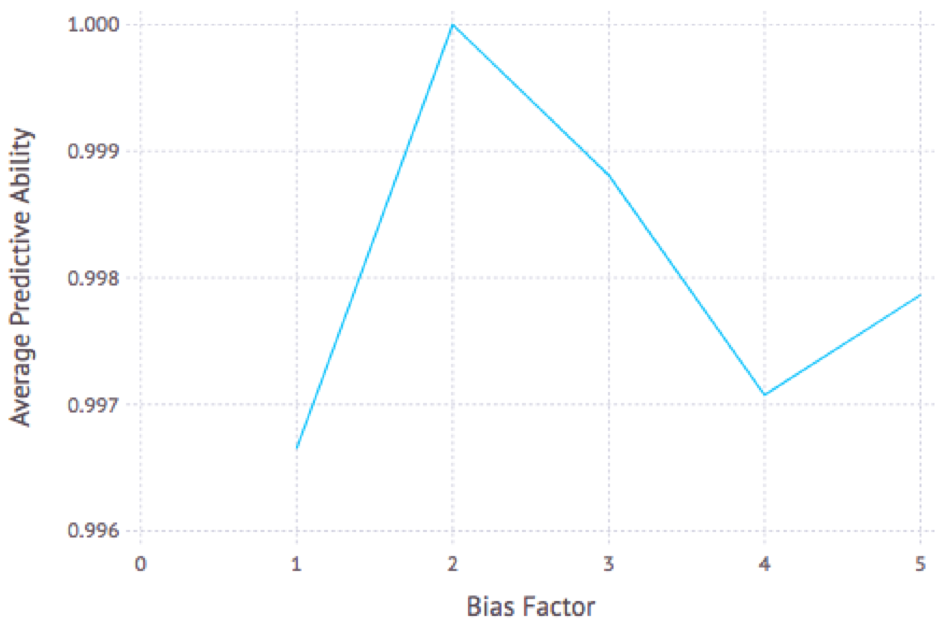
Figure 4: Parameter Selection for Training the Sequence Classifier



3.6 Results

As discussed in the previous sections, I have selected to build a three layer, fully connected neural net with a single hidden layer consisting of 11 neurons. All outputs will use a sigmoidal activation function, and I will elect to train the neural net for 5000 epochs with a learning rate of 0.05. As demonstrated previously in Figure 5 and discussed in section 3.5, this architecture does not seem to be sensitive to class imbalance, and has a demonstrated prediction accuracy of around 99% after cross-validating 10 times. To note, I have performed essential data quality control, as discussed in section 3.2.

Figure 5: Impact of Class Inbalance on Predictive Ability with Optimal Parameters, $N = 5000$, $\alpha = 0.05$



To predict on the unknown data, I will train my neural net of 5000 epochs using a learning rate of 0.05 over 20 iterations of cross validation where I select 90% of the data at random to act as a training set and leave 10% of the data out for intermediate testing. To note, because I did not observe a significant difference in my neural network's ability to predict as I increased class inbalance, I will use a factor of 5, meaning that there will be 5 times as many negative examples as positive examples – I hypothesize that this will give my neural net a better chance at learning discriminatory features than using a balanced test set. After training my neural net for these 20 rounds, I will then use the final weights to report the probability of each un-annotated sequence being a transcription binding site, where higher probabilities correspond to this sequence being more likely to be a binding site. To note, I am *not* thresholding the values to be 1 or 0 such that my overall performance can be evaluated with an ROC curve. My final results are available in the file `final_predictions.txt`.

4 Conclusions

Here I have introduced my implementation of a neural net framework for classifying types of genomic sequences. In particular, I developed a simple three layer neural net with a single hidden layer consisting of 11 neurons that can discriminate between transcription factor binding sites and neutral genomic sequences from only 17 base pairs of sequence. I have thoroughly benchmarked the performance of my neural net with respect to many different variables such as learning rate, number of epochs, number of neurons in the hidden layer, and training data class imbalance. In the end, I settled on the 11 neurons mentioned before along with 5000 epochs trained with a learning rate of $\alpha = 0.05$.

I also outlined my implementation of a autoencoder to find a low dimensional latent space for simple 8 dimensional one hot encodings. I reported on the impact of various training examples as well as the resulting latent dimensional representation looks like.

Future work for this neural network could focus on perhaps implementing a convolutional layer as well as supporting a 4×17 dimensional tensor as input. Previous work has illustrated the success of such convolutional neural networks for the purpose of classifying transcription factor binding sites (e.g. DeepSea, 2015) and thus would be an interesting improvement to the neural net described here. Notably, I report a near perfect classification accuracy upon cross-validation on 10% held out data, so it is unclear how much improvement could be gained using more sophisticated and involved neural net architectures.

5 Code Availability

All code used to generate figures as well as a package that supports a basic command line interface is available at `https://github.com/mattjones315/NeuralNet.jl`. The neural net code is located in `src/nn.jl` and the notebook used to generate all figures and benchmark the code is in the home directory under `benchmarks.ipynb`.