**Describe machine learning approach in detail**

To convert the DNA sequences into a format the encoder could use, I translated each base into a four-bit number ( A = 0001, T = 0010, etc). For each sequence, I converted each base to its corresponding number and concatenated them together, resulting in 68 digit long arrays from 17 bp long sequences. I chose this method over encoding the bases as single bit numbers (1, 2, 3, and 4) because I wanted to avoid the value of T being considered twice the value of A etc.

Given input in this format, along with a corresponding label (0 for negative, 1 for positive), my method first generates a neural net of the appropriate dimensions (number of weights equal to length of input times number of hidden nodes, output equal to dimension of labels, etc). The number of nodes in the hidden layer as well as the learning rate can be set as parameters. The initial weights are generated randomly.

Once the neural net has been initiated, it can be trained by iterating through forward and back propagation. The forward propagation takes the dot product of the input layer and the first set of weights to generate a value for each of the hidden nodes. I then apply a sigmoid activation function to convert this value to an output.

For back propagation …

I chose a sigmoid activation function because it thresholds my data between 0 and 1 which is useful since I want my output to be the probability of a class (binding site) which I can encode as 1.

**How was your training regime designed so as to prevent the negative training data from overwhelming the positive training data?**

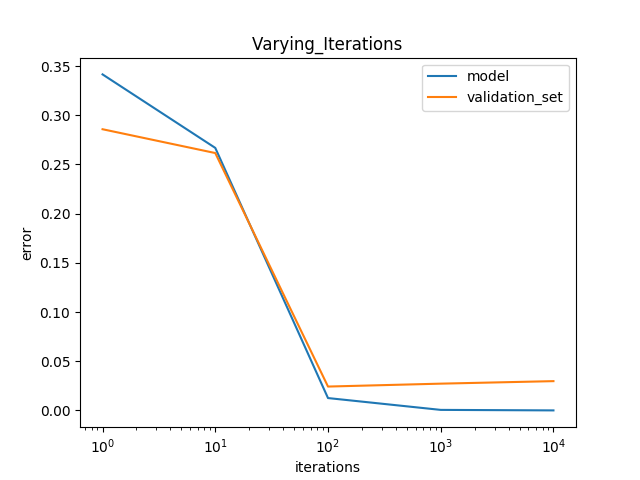
Because there are several orders of magnitude more negative sequence than positive sequence, I randomly sampled the negative sequences to get balanced classes. To so I randomly sampled an equal number of negative sequences and then randomly sampled a 17 bp stretch of each of these sequences. I removed any sequences that were an exact match to positives (although in principle this could result in unbalanced classes, in practice I never found an exact match and so don’t expect to lose more than 1 or 2 negative sequences at most).

To train I combined the positive (both given and reverse-complemented) and the downsampled negative data and generated labels for them (0 for negative, 1 for positive).

**What was your stop criterion for convergence in your learned parameters? How did you decide this?**

I decided to run my training function for a set number or iterations or until the error fell below a certain threshold. To determine the appropriate maximum number of iterations and error threshold I tested a series of training iterations and compared the model error and validation error. To avoid overfitting on one training set, this was done in conjunction with k-fold cross validation (see below).

The model error was defined as the mean squared difference between the labels and the final output. The validation error was defined as the mean squared difference between the prediction and the labels. The plot below shows how the model error and validation error changed with the number of training iterations.



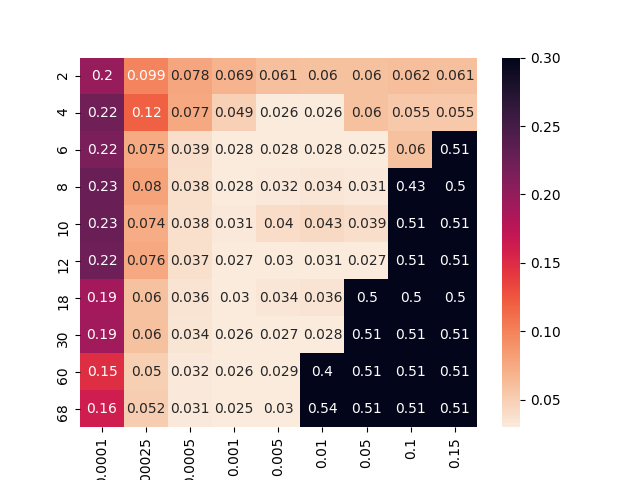
As I did not see an improvement in either error beyond approximately 1000 iterations I set this as my maximum to avoid overfitting the model to my training data. (I did not see the validation error increase at this point so I am confident the model is not overfitting). I set .01 as my error threshold as this was about the lowest I saw after running this process several times. In this way the model with either run for 1000 iterations (beyond which I do not expect any improvement, but before I expect overfitting) or it will stop if the model error gets as low as .01.

**Describe how you set up your experiment to measure your system's performance**

To test performance across a range of parameters, I used kfold cross validation. I first held out 10 percent of my data to test after selecting the optimal parameters.

I split the remaining 90% of my data into 5 random folds (resulting in about 100 observations in each fold). Each fold was used as the testing set while using the other 4 folds combined as the training set. The error (from the testing fold) was recorded for each set of parameters and averaged across folds. In this way, the neural net “sees” each fold an equal number of times and each fold is used once for testing.

I varied both the learning rate and the hidden layer size (number of nodes in hidden layer) in a nested loop and recorded the average validation error (average error from each fold combination) for those combinations of parameters. To visualize, I plotted the results as a heat map (y axis is size of hidden layer and x axis is learning rate).



As expected, very low and very high learning rates performed poorly. High learning rates cause the weights to update by such a large amount they may not converge. Low learning rates cause the weights to update by such a small amount, it would probably take many more iterations than my cutoff (1000)

There was a fairly wide range of hidden layer sizes that performed similarly well (6-30). Too few nodes may be unable to discriminate enough patterns to make accurate predictions. Too many nodes may result in overfitting of the data. This can be seen in the heatmap below where the error rate is very high when the number of nodes approaches the size of the input (68).

Based on the heat map, I selected a hidden layer size of 12 and a learning rate of .01 for my final parameters (the exact values varied slightly run to run, but this combination was always amongst the lowest). To confirm that these parameters worked well in general, I used the held-out data. I trained a neural net with all 90% of my training data using the selected parameters and then predicted the value of the 10% of held out data. By comparing it to the known labels, I was able to calculate an AUC of 98%.