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.

I tested the tanh function as well. Since the tanh function exists between -1 and 1 it will map negative inputs more strongly negative. This seems useful since I want to separate my two classes of data as clearly as possible.

How was training regime designed

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).

Stop Criterion

I decided to run my training function for a set number or parameters 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. This was done by training on 70% of my data and testing on the other 30% of the data. 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 split, train, validate process was repeated 10 times for each number of iterations tested and the error was averaged.

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 .02 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 .02.

Performance

I measured performance as above (70/30 training validation split, repeated 10 times and averaged). After determining the maximum iterations, I only considered the validation error and not the model error. I also held out 10% of the data before any splits, to be used for validating the best parameters once they were found.

Because 10 random splits could potentially retrain on the same subset of data multiple times, it would be better to use k-fold cross validation. This would involve splitting the data into k bins and using each bin as the testing set in turn (while combining the remaining bins as the training set). This would ensure that each trained neural net “saw” all of the data in the training set an equal number of times. I had trouble finding a solution for combining all of the training bins for k-fold cross validation so used the bagging method above.

I varied both the learning rate and the number of nodes in the hidden layer in a nested loop and recorded the average validation error (average error from each of the 70/30 splits) for those combinations of parameters. 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)

Too few nodes may be unable to discriminate enough patterns to make accurate predictions. Too many nodes may result in overfitting of the data. In the extreme case, if the number of nodes was the same size as the input later, they would just memorize the inputs.

Surprisingly, there was a wide range of hidden layer size that produced very similar errors. I tested several of the lowest combinations of node size and learning rate and found that they also had similar errors on the held out data.