1. Implement a feed-forward, three-layer, neural network with standard sigmoidal units. Your program should allow for variation in the size of input layer, hidden layer, and output layer. You will need to write your code to support cross-validation. We expect that you will be able to produce fast enough code to be of use in the learning task at hand. You will want to make sure that your code can learn the 8x3x8 encoder problem prior to attempting the Rap1 learning task.

Code for the creation and training of the neural network can be found in train\_nets.py. The network is capable of solving the auto-encoder problem:



1. Set up the learning procedure to allow DNA sequences as input and to produce an output of the likelihood that an input is a true Rap1 binding site. Describe the machine learning approach in detail. This will include, for an ANN, a description of the network structure of your encodings of inputs and output.

DNA sequences were converted to binary values using one-hot encoding as follows:

A 🡪 1000

C 🡪 0100

T 🡪 0010

G 🡪 0001

The ANN takes as input a matrix in which each row represents a potential binding sequence, where each sequence is a list of 18 integers indicating the base pair at that position plus a bias vector. These are multiplied by a set of weights to create a three-node hidden layer, which is then fed forward to an output value by multiplying by a second set of weights. The output represents the probability that a DNA sequence is for a RAP1 binding site.

Tune-able hyper-parameters:

* inS, outS, hS: # of nodes in the input, output, and hidden layers, respectively
* actFunction: activation function (“sigmoid”, “ReLU”, or “tanh”)
* batch\_size: number of samples by which to calculate gradient descent of weights
* epochs: number of times to sample the test data in batches
* metric: metric by which to minimize cost (“mse” or “roc\_auc”)
* learningRate

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

To prevent issues of class imbalance, the training function samples from the full dataset in sizes of batch\_size, randomly selecting (with replacement) from positive and negative examples with equal probability. Therefore, the model sees negative and positive examples in roughly equal proportions. This will prevent the model from assigning “0” probability to everything to maximize predictive scores; however, this leads to a higher risk of overfitting.

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

A grid search was performed to learn the ideal option for combining each of the five tunable parameters, with a total of ~2000 possible combinations:

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

One fifth of the data was held out of the training for these parameter combinations, and then tested against the created model and scored based on classification accuracy. Therefore, instead of converging upon a parameter space, the best possible combination of the above options was selected based on accuracy. The scatterplot below shows accuracy plotted by activation function and colored by optimization metric options, with highest and lowest accuracy parameters labeled:

This test was performed using three different random seeds, and it was confirmed that the ideal parameter set was consistent:

1. What set of learning parameters works the best? Please provide sample output from your system.

Ideal parameters:

With a test/train split of 20/80, the accuracy of the model is .

Figure showing the cost/score of the model over the epochs:

1. What are the effects of altering your system (e.g. number of hidden units or choice of kernel function)? Why do you think you observe these effects?
2. What other parameters, if any, affect performance?