Final Project

George Hartoularos | March 23, 2018

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.**

   I implemented a three-layer neural network that uses a sigmoidal activation function. The network allows for a variable number of input, hidden and output layers, and it is able to complete the auto-encoder problem in a matter of seconds with 10000 iterations. I organized the algorithm into separate functions that pass a dictionary (called `parD`, for "parameter dictionary") to each other, which stores all the parameters of the model. The dictionary has a tuple as keys, with each tuple having two elements, the first being an integer marking the layer, and the second being a character marking what type of value (e.g. 'w' for weight, 'a' for activation). To run the auto-encoder, execute:

   ```
   $ python neuralnet.py auto 10000
   ```

2. **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. You will want the output value to be a real number no matter what method you are using, since your performance on the blind test set will be judged based on the area under an ROC curve. So, please DO NOT threshold your output to produce a binary value.**

   In order to represent my sequences as a numerical value (so that it can be inputted into the algorithm), I used the "one hot" encoding tactic. This takes an arbitrary alphabet of *n* letters used to encode a sequence and converts each letter to a string of binary that is *n* digits long. The string is made up of all but one zeros; a one occupies a unique position which encodes the letter of the alphabet (see Table below).

Nucleotide	One-hot Encoding
A	1000
C	0100
G	0010
T	0001

I used a similar architecture as was used for the autoencoder, but now using 68 neurons in the first layer (as this corresponds to exact number of digits I needed to encode a 17-base-long sequence with 4 digits each), half that (34) in the hidden layer, and only 1 neuron in the last layer. The one neuron corresponds to a yes/no degree of accuracy as to whether a sequence is representing a binding site or not; it was not binarized.

3. **Design a training regime that will use both positive and negative training data to induce your predictive model. Note that if you use a naive scheme here, which over-weights the negative data, your system will probably not converge (it will just call everything a non-Rap1 site). Your system should be able to quickly learn aspects of the Rap1 positives that allow elimination of much of the negative training data. Therefore, you may want to be clever about caching negative examples that form "interesting" cases and avoid running through all of the negative data all of the time.**

   To train my model, I read in the positive and negative sequences. Because the length of the negatives greatly exceeded the length of the positives, I selected a random 17-nucleotide sequence. I then ensured that none of the positive sequences were in the negatives, and if so, removed that negative from the list. Additionally, because the *number* of negatives greatly exceeded that of the positives, I also chose a random 137 negative sequences to achieve class balance. I did not have a stop criterion for the error: rather, I just ran the simulation for a large number of iterations.

4. **Given your learning system, perform cross validation experiments first to see whether it is working, and then see how well your system performs.**

   Cross-validation was performed using sklearn's `model_selection.Kfold` function, which streamlines the process of dividing up the training set into k number of subsets, training on $\frac{k-1}{k}$ % of the data and testing on the remaining $\frac{1}{k}$ %, and repeating for every subset for a total of k times. For each iteration, my AUC is 1.0000, shown up to 4 decimal places. Although I did not systematically examine the effect of learning rate, number of neurons, and other algorithmic choices, simply varying the learning rate from between 1 and 100, and changing the number of neurons from 10 to 40 in the hidden layer, did not change the results. The following is a sample output of the bias and weight matrices from one validation of the Kfold process.

```
(0, 'B'): array([[-0.0100865 ,  0.00086164, -0.04030826, -0.11518993,
0.17229163, -0.00575901, -0.19032978, -0.00332879,  0.09700079,
0.1685012 , -0.15647526,  0.19700218,  0.01775047, -0.00621646, -
0.08099231, -0.12593245, -0.06853399, -0.03032984,  0.13256   , -
0.01630157, 0.00702103,  0.00175656, -0.04709186,  0.25186093, -
0.18008071, -0.00273117, -0.01159603,  0.01411976,  0.27045477,
0.08036608, 0.00970258,  0.01028492,  0.1971074 , -0.14879299]]),

(0, 'W'): array([[-0.04891156, -0.0132821 ,  0.05658621, ..., -
0.02800386, 0.10875839,  0.10456691],
       [ 0.00595183, -0.01273361,  0.00679359, ...,  0.00034226,
         0.02705682, -0.04619095],
       [ 0.00611287,  0.01736968, -0.02579516, ...,  0.00644024,
         0.07232757, -0.06486682],
(1, 'B'): array([[-0.0894931]]),
(1, 'W'): array([[-0.27518214],
       [-0.13154733],
       [ 0.52949859],
       [ 1.0333244 ],
       [-1.30424905],
       [ 0.18014096],
       [ 1.35778207],
       [-0.00968851],
       [-1.04247423],
       [-1.35574262],
       [ 1.13167246],
       [-1.55757078],
       [-0.48148915],
       [ 0.35199707],
       [ 0.811198  ],
       [ 1.03438818],
       [ 0.69363203],
       [ 0.53702622],
       [-1.13992362],
       [ 0.08731023],
       [-0.46185299],
       [-0.13732636],
       [ 0.69938071],
       [-1.63799219],
       [ 1.25426971],
       [-0.24123227],
       [ 0.41003885],
       [-0.41596619],
       [-1.76359626],
       [-0.90715717],
       [-0.48747579],
       [-0.25724339],
       [-1.31606973],
       [ 1.17300207]])
```

5. See predictions.txt file.