Final Project

BMI 203

Winter 2018

github repo: https://github.com/christacaggiano/-neural-net

Assignment- Distinguish real binding sites of a transcription factor (RAP1) from other sequences.

Part 1 - autoencoder

Feed forward 3 layer neural network with standard sigmoidal units.

8x3x8 autoencoder use:

```
from main import autoencoder
matrix_size = 8
print(autoencoder(matrix_size))
```

```
Optimization terminated successfully.
         Current function value: 0.000004
         Iterations: 123
         Function evaluations: 202
         Gradient evaluations: 5666
         Hessian evaluations: 0
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

Part 2 - Learn transcription factor binding sites

I chose to continue to use my artificial neural network (ANN) to learn transcription factor binding sites.

encoding

In order to feed DNA sequence information in my ANN, I used one-hot encoding. I assigned a value to each of the nucleotides (i.e `d = {"A": 0, "T": 1, "C":`

`2, "G": 3}`) to integer encode the sequence information. Since this encoding suggests an implicit ranking of the nucleotides, like `G==3 > A==0`, this overweights my neural network towards G's. For the purpose of this assignment I assumed that all the nucleotides are more or less of equal importance (in future work, the distribution of nucleotides in the positive and negative test data could be examined), one-hot encoding transforms this integer encoding to something more egalitarian. Using `sklearn.preprocessing import OneHotEncoder` the 17 nucleotides in the binding motif sequence were turned into a 17x4 array. If the sequence was **ATCG**, the encoding could look like:

```
#  A   T   C   G
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

Since I have 137 positive test values, the resulting data would be of shape `(137, 17, 4)`. Since a 3-D array is difficult to deal with, I chose to flatten my arrays, taking the `(17, 4)` matrix encoding each sequence into an `(68, 1)` matrix. In the example above, our matrix would become

```
[ 1.  0.  0.  0.  0.  1.  0.  0.  0.  0.  1.  0.  0.  0.  0.  1.]
```

I chose to break up the negative training label into 17-mers, to match the positive data. I started from the beginning of each fasta file, and discarded remaining bases that did not evenly fit into 17-mers. Future work could involve computing a training set that encompasses all possible permutations of 17-mers.

neural network design

My ANN takes in a 2D array. I chose to set the hidden layer size at 3, as qualitatively, adding more nodes did not seem to improve the quality of my predictions. Future work could be creating a deep neural network with more layers, however, I found that a single layer seemed sufficient for the task. I began by initializing random weight values in the network. The network structure takes the input nodes, uses the first weights to computed the second 'hidden layer.' I used the standard sigmoidal activation function to determine the amount each node contributed to the output layer. Then, the weights of the final layer, also initialized randomly, were multiplied by this activation matrix. A final use of the sigmoid function gave the output as a value between 0 and 1, the probability that the data was a true binding site.

parameters

- Convergence: my optimization was allowed to proceed for 7,000 iterations with a tolerance of 1e-8. I chose these convergence criteria as they allowed very precise calculation within the compute power of a standard Macbook pro.

- Learning rate: a learning rate of 0.01 was chosen as it was a fairly large learning rate that produced reasonable accuracy. In the future, the learning rate could be optimized through cross-validation to ensure I picked the best learning rate for my problem

training

For training, a batch gradient optimization was used. Cost was calculated as the square distance between the true and predicted values. All the errors were summed. The gradient was calculated as the change in this cost function. I used scipy gradient optimizer to update the weights of my NN in order to minimize the cost. I found that Newton conjugate gradient method performed the most consistently on my data.

To prevent negative training data from overwhelming the positive data, I just took a random subsample of the negative data to train and a random subsample to test. I found ~500 samples provided good results. This is a fairly naive solution. In the future I would like to cluster the negative training data into particular classes thta could be sampled to better train the network

validation

Random samples of the data were chosen for training and testing to evaluate the model's performance. Several iterations of this was performed to achieve at least a sense of confidence in my model.

I found that the parameter that my model was most sensitive to was the optimization method. I found that BFGS was an extremely common optimization method, but it performed terribly on my data, especially my autoencoder. It created many false positives. In contrast, BFGS caused a high false negative rate in my DNA data, something I do not quite understand. Nelder-mead optimization often failed to converge. Newton-CG converged very quickly and produced really consistent results, which made me favor it over other methods. I didn't use a hessian.

Sample output for `hidden_size = 3, num_subsample=500` and 50% of the data reserved for training.

```
Optimization terminated successfully.
         Current function value: 0.000004
         Iterations: 23
         Function evaluations: 27
         Gradient evaluations: 199
         Hessian evaluations: 0
```

```
[[ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
 [ 0.    ]
[ 0.   ]
[ 0.   ]
[ 0.001]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 0.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 0.983]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 0.999]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 0.996]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 1.   ]
[ 0.999]
[ 1.   ]
[ 1.   ]
```

```
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 0.923]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 0.999]
 [ 0.637]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 0.978]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]
 [ 1.   ]]

Process finished with exit code 0
```

Sources consulted:

- Stephen Welch, Neural Network Demystified
- Suriyadeepan Ram, The Principle of Maximum Likelihood
- @giantneuralnet, Beginner Intro to Neural Networks
- Jon Como, Flowers, a simple neural net tutorial
- Sebastian Rudder, An overview of gradient descent algorithms
- Erik Lindernoren, ML From Scratch