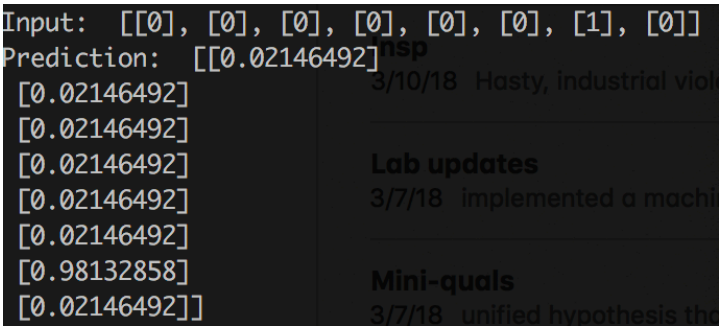
1. Implement a feed-forward, three-layer, neural network with standard sigmoidal units. Your program should allow for variation in the size of input layer, hidden layer, and output layer. You will need to write your code to support cross-validation. We expect that you will be able to produce fast enough 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:



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. 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 17 encoded nucleotides 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 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

3. 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. Additionally, the cost function for gradient descent uses the area under the precision-recall curve to discourage false negatives (as well as false positives). However, both of these measures lead to a higher risk of overfitting.

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

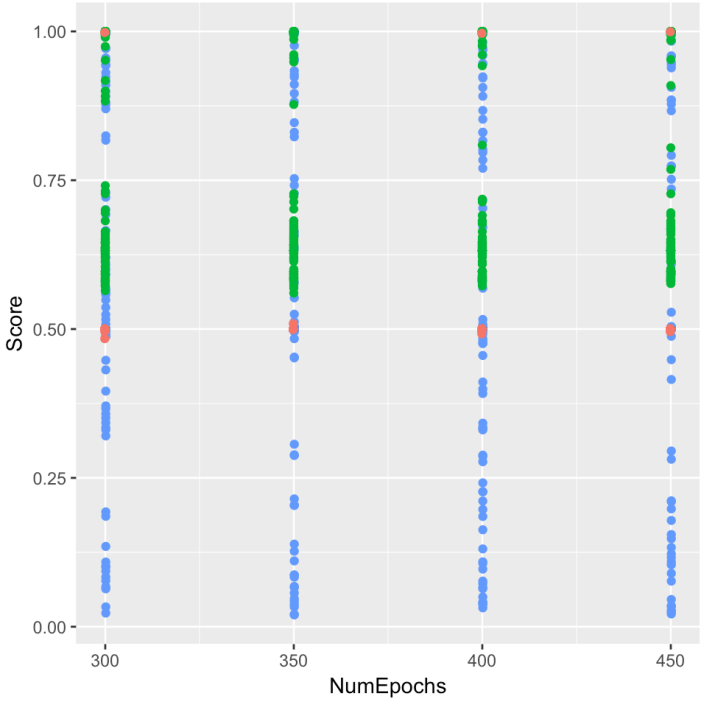
A grid search was performed to learn the ideal option for combining each of the five tunable parameters, with a total of ~2000 possible combinations:

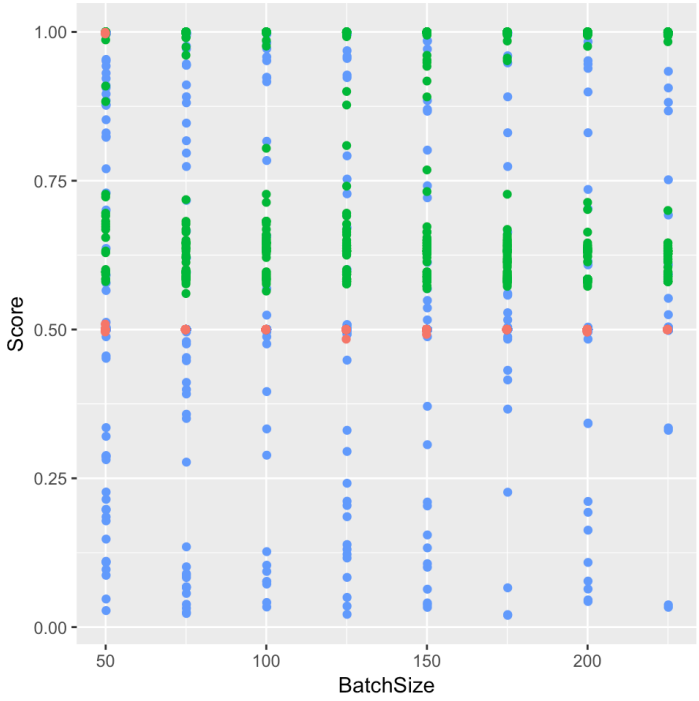
```
# Parameter grid search
act_opts = ["sigmoid","ReLU","tanh"]
epoch_opts = np.arange(300,500,50)
batch_opts = np.arange(50,250,25)
metric_opts = ["roc_auc", "mse"]
LR_opts = {"sigmoid": np.arange(.20,.80,.05),\
           "ReLU":   np.arange(.0002,.0008,.00005),\
           "tanh":   np.arange(.002,.008,.0005)}

for actFunction in act_opts:
    for epochs in epoch_opts:
        for batch_size in batch_opts:
            for metric in metric_opts:
                for learningRate in LR_opts[actFunction]:
```

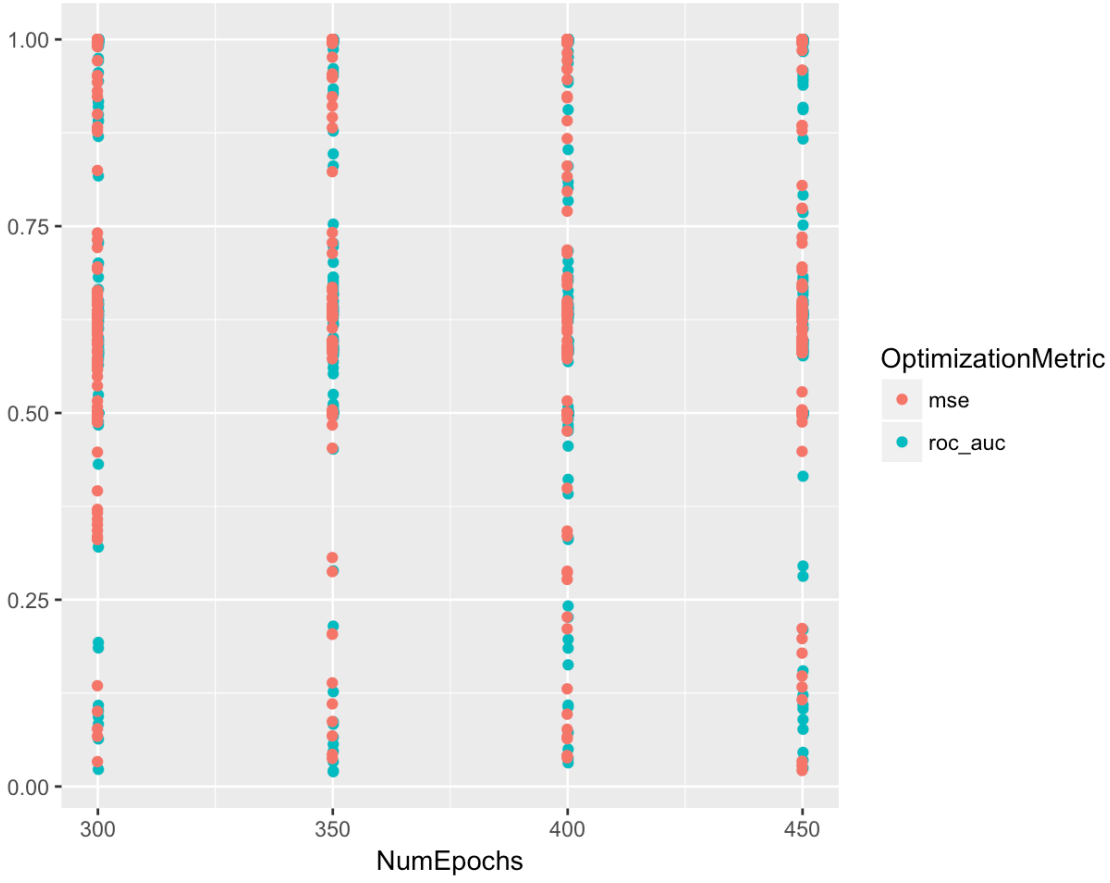
5. Describe how you set up your experiment to measure your system's performance.

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 scores. Therefore, instead of converging upon a parameter space, the best possible combination of the above options was selected based on the area under the precision-recall curve (AUROC). The scatterplots below show AUROC plotted and colored by

various parameter options. Overall, it seems that the sigmoid activation function is the clear best choice, but batch size, number of epochs, and choice of optimization metric required averaging over several tests to determine the optimal parameters (see #6).







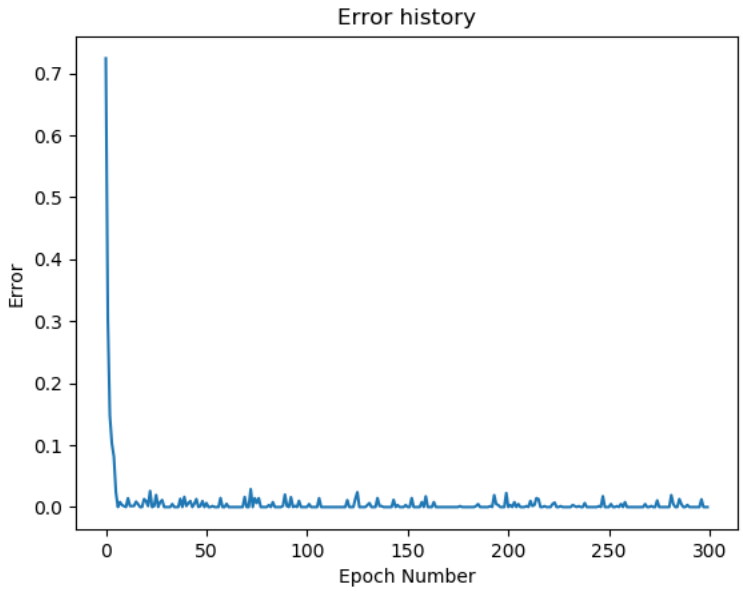
6. What set of learning parameters works the best? Please provide sample output from your system.

   Ideal parameters:

```
Best parameters:  {'Input Size': 18, 'Output Size': 1, 'Hidden Layers': 3, 'Activation Function'
: 'sigmoid', 'Number Epochs': 300, 'Training Batch Size': 50, 'Optimization Metric': 'roc_auc',
'Learning Rate': 0.44999999999999996}
AUROC score:  1.0
```

   With a test/train split of 20/80, the accuracy of the model is 1.0, which is probably indicative of overfitting.

   When the model is trained using these ideal parameters and the full training dataset, the following figure shows the error history of the model score (given by 1 - AUROC) at each training epoch.

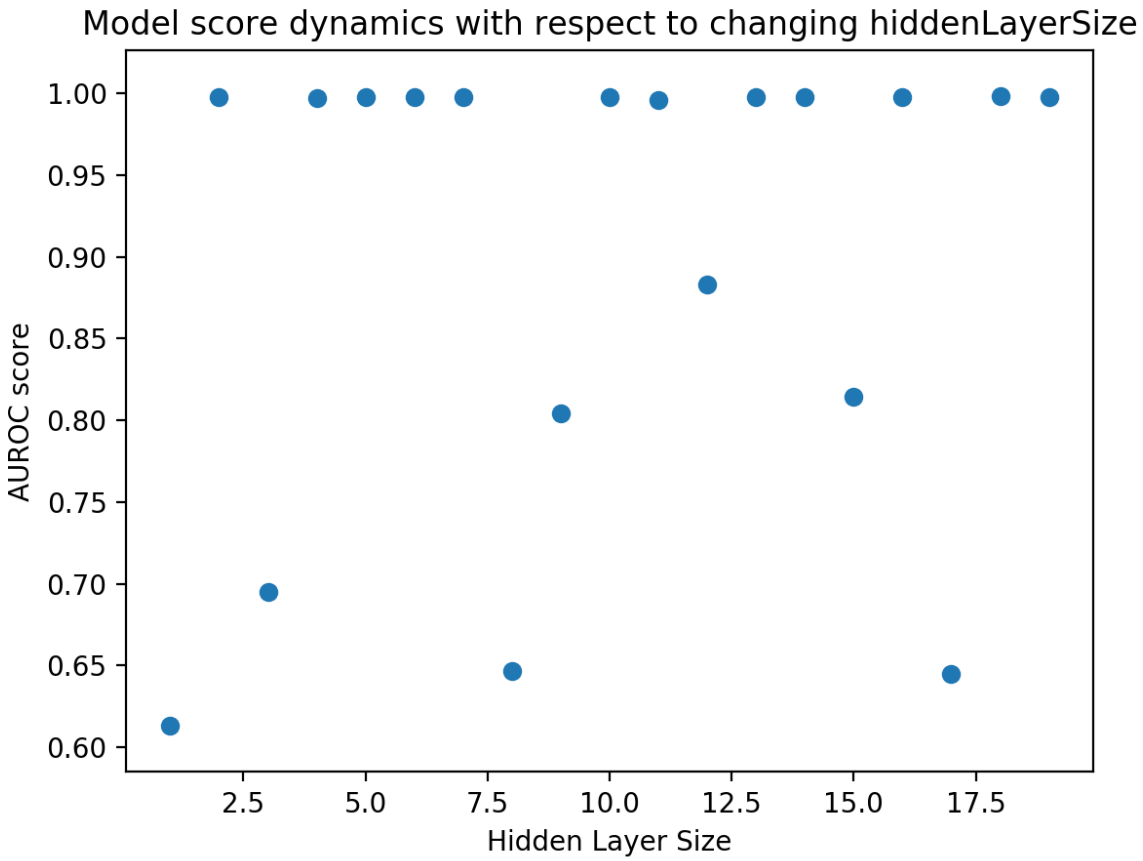


   The model seems to learn very quickly, within just a few epochs, and then stochastically explores areas that may lead to higher error. Perhaps this is an indication that fewer epochs should have been used, and finding an average minimum at 300 (the chosen parameter) was random chance.

7. 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?

   Different parameters were explored in #5 – notably, the ReLU activation function's score is more consistently 0.5 than 1.0. The tanh activation function yielded a wide spread of prediction scores, ranging from nearly 0 to nearly 1.

Altering the number of hidden nodes showed an interesting pattern in which most sizes yielded high prediction scores, but others were much lower. I suspect that hidden layers of these sizes were prone to overfitting and thus performed poorly on the held-out test data.



8. List of sequences can be found in output/predictions.txt