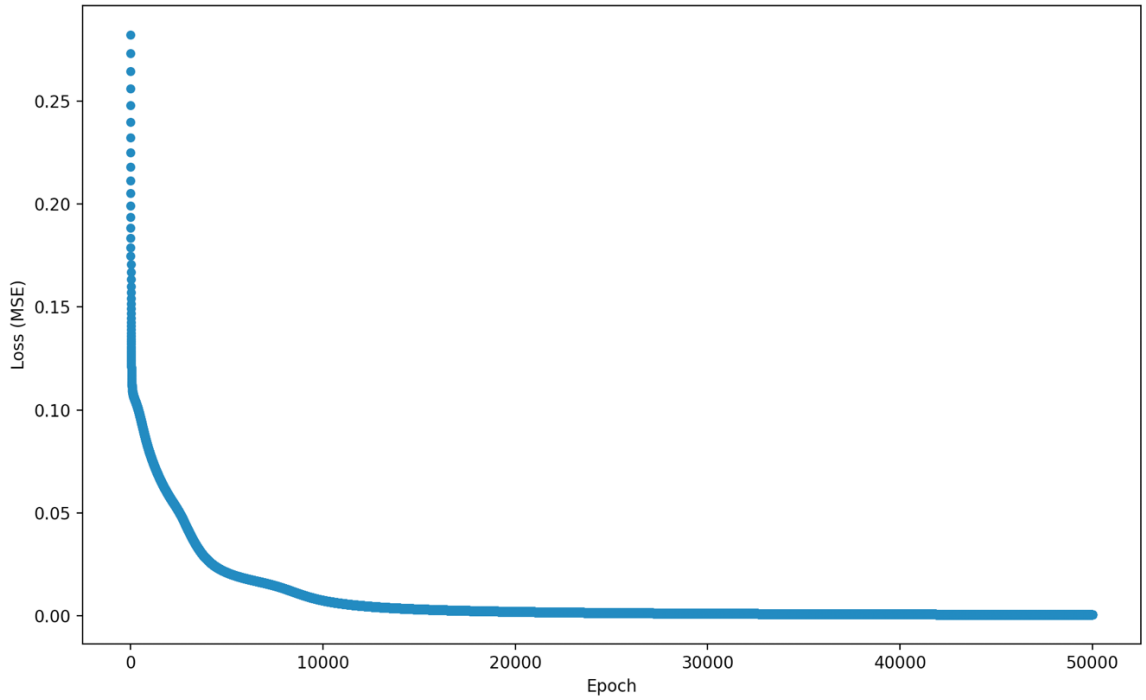
Ryder Easterlin
March 19, 2021
BMI 203: Project 3

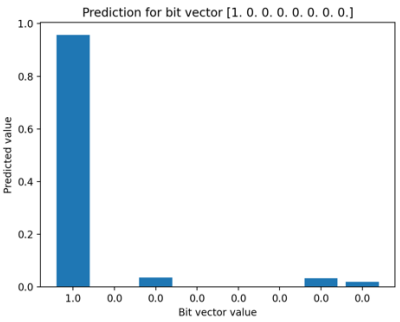
Part 1: Autoencoder implementation (10 points)

1) **Implement a feed-forward, neural network with standard sigmoidal units. Your implementation should accept a vector as input, be able to adjust network weights by backpropagation, and report the average loss per epoch. It should also allow for variation in the size of input layer, hidden layers, and output layer. We expect that you will be able to produce fast enough code to be of use in the learning task at hand.**
   a. **To confirm that your implementation functions correctly, demonstrate its ability to correctly solve the 8 x 3 x 8 autoencoder task. Specifically, set up an autoencoder network consisting of an input layer (8 nodes), a hidden layer (3 nodes), and an output layer (8 nodes), all with sigmoidal units.**

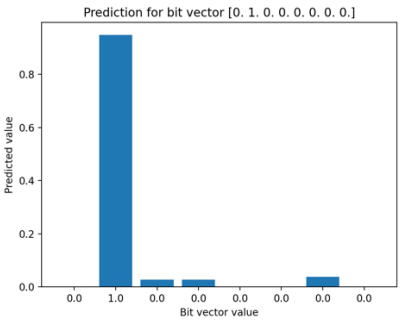
To prove my implementation's ability to learn an autoencoding of 8-bit vectors, I will train the autoencoder with the above architecture on all possible 8-bit vectors with a single activation (this is the identity matrix). I will plot the loss curve and display my autoencoder's final prediction for each bit vector.

I will train for 50000 epochs (I know, that's a lot) with a learning rate of 0.2. I will train for this many epochs because the network only sees 8 samples per epoch, and I am only trying to demonstrate that the network learns without regarding overfitting or anything.

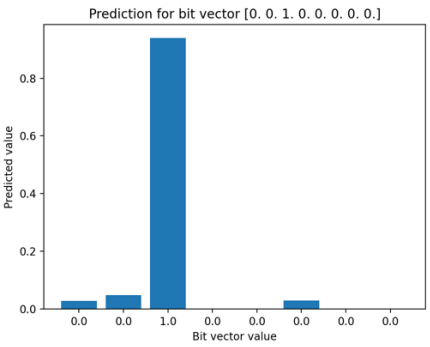




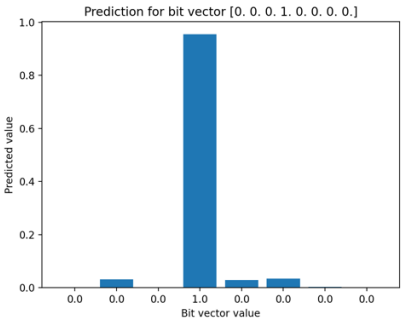
1



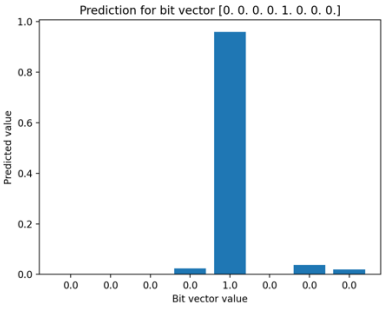
2



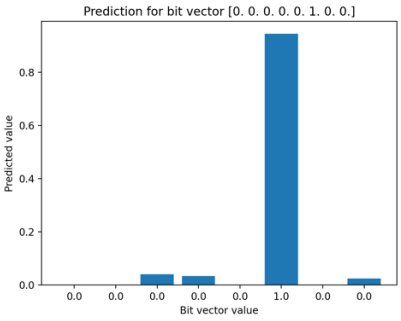
3



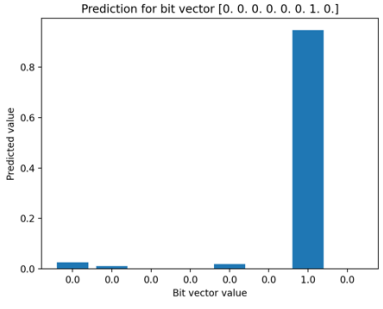
4



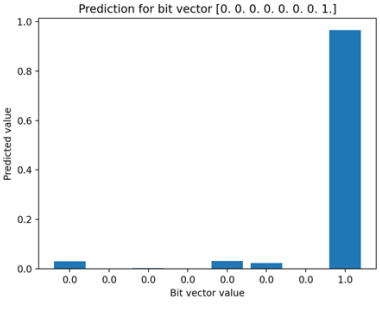
5



6



7



8

I plotted the predicted bit-vector for each true bit-vector above. Clearly, the autoencoder finds a good encoding for each single bit bit-vector in the 8x8 identity

matrix. Each prediction puts a value of over .9 in the correct index of the corresponding input.

Part 2: Adapt for performance, and develop training regime (10 pts)

2) **Set up a procedure to encode DNA sequences (Rap1 binding sites) as input for your neural network. Consider how your encoding strategy may influence your network predictions.**
    a. **Describe your process of encoding your training DNA sequences into input vectors in detail. Include a description of how you think the representation might affect your network's predictions.**

I opted to one-hot encode the DNA sequences. Specifically, each base in the sequence is encoded as a 4-bit vector with 1 activation, and the position of the activation indicates which base it is. Adenine is encoded as [1,0,0,0], thymine is encoded as [0,1,0,0], cytosine is encoded as [0,0,1,0], and guanine is encoded as [0, 0, 0, 1]. Therefore, each sequence is initially encoded as a vector of 17 4-bit vectors. This 2d array is then flattened out so each sequence is encoded as 68 bits (17x4). One potential drawback of this encoding scheme is that it quadruples the feature space of the learning task, necessitating a wider network which probably requires more epochs to train, compared to integer encoding of the bases. However, one-hot encoding is usually better for non-ordinal categorical variables, such as DNA bases, so I will stick with that rule of thumb.

Since the negative sequences are provided at variable lengths, I will break up each negative sequence into all of its constituent 17-base pair subsequences and include each of these subsequences from each negative sequence in one large batch. Duplicates in this batch are removed, so each 17-base pair sequence is unique.

3) **Design a training regime that will use both positive and negative training data to train your predictive model. Note that if you use a naive scheme here, which overweights the negative data, your system will probably not converge (it will just call everything a non-Rap1 site).**
    a. **Describe your training regime. How was your training regime designed so as to prevent the negative training data from overwhelming the positive training data?**

I first remove any 17-mers from the negative examples that also appear in the positive examples. This leaves me with a total of 137 positive sequences and 2,982,612 negative sequences. I then (randomly) divide each class into 90% training/10% testing sets. I therefore have 123 positive training cases, 2,684,351 negative training cases, 14 positive testing cases, and 298,261 negative testing cases. The model is then fit with the training cases of each class. Importantly, in each epoch the model is only trained on 123 negative examples that are randomly selected from the pool of 2,684,351, and the full set of 123 positive training examples. Therefore, in each epoch the model sees the same number of positive and negative cases (123 of each). This method ensures that

the negative training data does not overwhelm the positive data, while also taking advantage of the large number of unique sequences in the negative training data. Also, this is like a combination of mini-batch and full-batch gradient descent, but stratified by class (negative and positive).

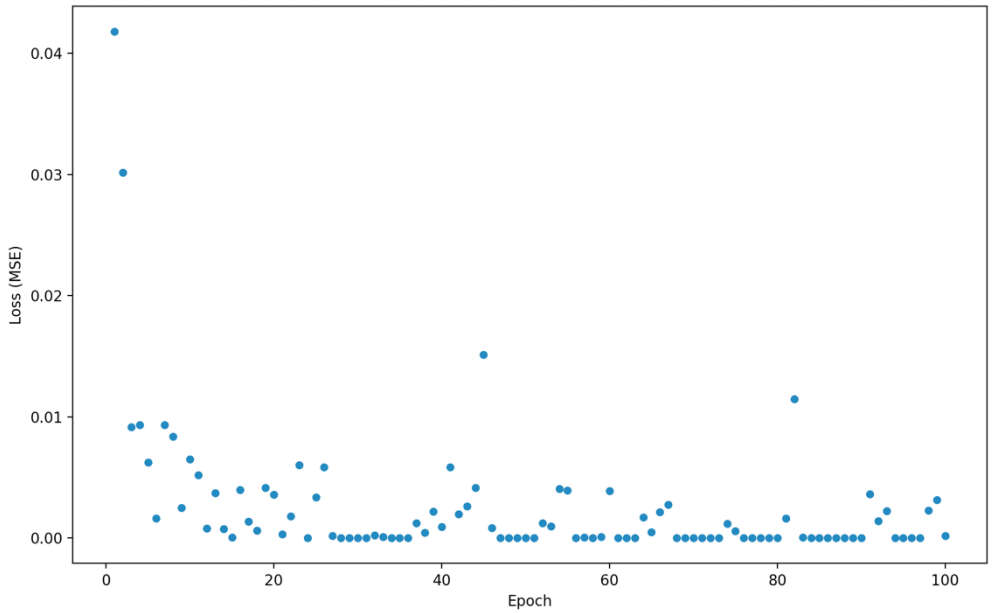
I then trained for 100 epochs (rationale explained below).

4) **Modify your implementation to take as input positive and negative examples of Rap1 binding sites (using your encoding from Q2) and produce an output probability between [0 - 1.0] indicating classification as a binding site (1.0) or not (0.0) . Select a network architecture, and train your network using the training regime you described in Q3 on all the data.**
    a. **Provide an example of the input and output for one true positive sequence and one true negative sequence.**

Sequence	**True Label**	**Prediction**
ACACCCACACCCCTCAT	1 (Positive)	0.99974547
AAACTGAAGGAGTTTTT	0 (Negative)	3.17731337e-11

    b. **Describe your network architecture, and the results of your training. How did your network perform in terms of minimizing error?**

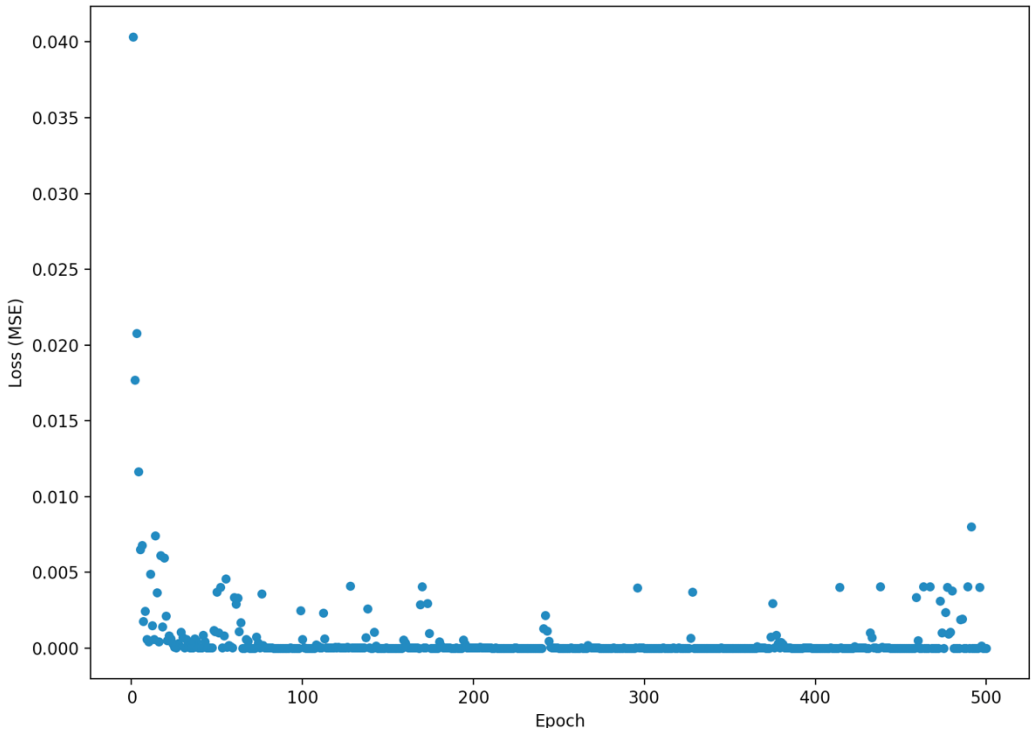
I toyed around with a couple architectures, loss functions, and activation functions, and found that a simple single hidden layer architecture worked really well. Specifically, the input layer was 68 nodes, followed by an 8-node hidden layer with ReLu activation, followed by the single output node with sigmoid activation. I also found that mean-squared error as a loss function resulted in better performance than binary cross-entropy. In general, it learned really well and I've plotted the training loss curve below over the 100 training epochs. Clearly, it minimizes training error fairly quickly. Because I used a quasi mini-batch gradient descent training scheme, we see some variation in the loss even after the model has converged. The broader trend is that the training loss decreases over the training epochs, which is what we are looking for.



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

Ideally, I would like to create a validation set that I test the network on after every epoch, and stop training when the loss on the validation set stops declining for a certain number of epochs in a row (3, for example). However, I opted to not do this due to the scarcity of positive examples that I have access to. I simply wanted to include as many of them as possible in the training phase, and having a validation set would remove some of the training examples.

So, I instead opted to simply train the network for far more epochs than necessary, and inspect the training loss curve to see where the loss stops declining rapidly. Of course, the training loss will always decline until the training data is virtually (or actually) memorized, so this inspection is to some degree a judgement call for when we stop seeing large decreases in loss. Below, I show the loss curve over 500 epochs. The loss trend stops declining noticeably around 100 epochs, so I decided on a training length of 100 epochs moving forward.



5) **Evaluate your model's classification performance via k-fold cross validation.**
   a. **How can you use k-fold cross validation to determine your model's performance?**

k-fold cross-validation is often used to assess model performance if enough data is present (which I'll discuss in b), since it tends to give a less biased assessments of model performance than a simple training/testing dataset split. Essentially, k mutually exclusive testing datasets (folds) are constructed from the main dataset, and for each fold a model is trained on the non-testing data and tested on the testing data. The testing results are then aggregated across these k folds.

k-fold cross-validation is typically thought to perform best with regards to the bias-variance tradeoff, since it is much less biased than a train/test split, but results across folds are typically not highly variant, as is the case with leave-one-out cross-validation.

Because the positive examples are so far outweighed by the negative examples in this case, it will be necessary to fold the positive and negative examples separately. This way, each fold should get roughly the same number of positive and negative examples.

   b. **Given the size of your dataset, positive and negative examples, how would you select a value for *k*?**

There is never a perfect answer to this question, even for class-balanced large datasets (which isn't even the case here). The choice of k will always be at least somewhat arbitrary, and people almost always choose a value between 5-10.

As mentioned above, there are relatively few positive examples relative to the amount of total data, so my choice of k will be based off of the number of positive examples more than anything. I will consider a choice of k=5 or k=10. In the case of k=10, each testing fold will have 13-14 positive examples, and in the case of k=5 each testing fold will have 27-28 positive examples. Considering that the feature space is 68 bits, both of these amounts of positive testing examples is insufficient, since there are more features than observations. However, the amount of positive testing examples for k=5 is less insufficient, so I will opt to use k=5 for my k-fold cross-validation.

c. Using the selected value of k, determine a relevant metric of performance for each fold. Describe how your model performed under cross validation.

Note: folding the nearly 3,000,000 negative 17-mers is really slow, so for the sake of testing this I randomly subset the negative data to 500,000 samples prior to folding. So, each of the 5 negative sample folds has 400,000 training samples and 100,000 testing samples. These negative folds are then grouped with the positive folds to make the final folds. I used the same training protocol as above (100 epochs, same architecture)

AUROC (AUC) is generally a good metric of the performance of classification models, so I will use that to compare and aggregate performance across folds. I will show the AUC below for each fold, and averaged across the five folds

Fold	**AUC**
1	0.9999396296296296
2	0.9999677777777778
3	0.9999551851851852
4	0.9999135714285715
5	0.9999214285714285
Average	0.9999395185185185

These are the most ridiculously high AUCs I've ever seen, indicating that the difference between binding sites and non-binding sites is pretty easily learnable. I'd actually be

interested to see how logistic regression performs on this task because this is kind of a piece of cake for a neural net, it seems.

Part 4: Extension (required but graded generously) (5 points)

Try something fun to improve your model performance! This should include implementation of alternative optimization methods (particle swarm, genetic algorithms, etc), you can also optionally add changes in the network architecture such as modifying the activation function, changing the architecture, adding regularization etc. For this section, we want to see a description of what you want to try and why. As long as we have this, and some effort towards implementation, you will get full points.

Okay so I think that performing PSO or genetic algorithm to optimize a network that already has an average AUC of 0.9999395185185185 is kind of overkill. Most networks are going to work really well at this learning task, and any AUC above what we are already getting could very well just be attributable to randomness.

Instead, I will implement dropout and L2 regularizations. I will then compare how the same model (architecture described above) performs with these regularizations implemented in the single hidden layer, and without. I will use dropout rates between .05-.5 (iterating by .05), and lambda values between .0005-.005 and display results for the best performing value of lambda/dropout rate for each regularization type.

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

System output:

```
BEST DROPOUT (rate, AUC): (0.3, 0.9998791190476191)

BEST L2 (lambda, AUC): (0.0005, 0.9999417486772486)
```

Regularization	**Average AUC (5-fold CV)**
Dropout (p=.3)	0.9998791190476191
L2 (lambda=0.0005)	0.9999417486772486
None	0.9999395185185185

These are the results of testing the different regularization techniques with 5-fold cross-validation. All three regularizations (Dropout, L2, and None) in the network result in very high AUCs, with L2 regularization slightly beating no regularization, which itself slightly beats dropout. Therefore, moving forward I will use L2 regularization with lambda = .0005 in the hidden layer of my network.

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?

When I was playing around with network architectures earlier, the shape of the network, activation functions used in each layer, and choice of loss function all affected the network's performance. For any given learning task, there will be an optimal network structure, loss function, activation functions, etc. All of these hyperparameter choices affect the way that the network learns the underlying signal in the data, and therefore some combinations of parameters work better than others. Therefore, hyperparameter optimization is usually necessary for harder learning tasks. However, it isn't necessary when I'm getting such a high AUC on a relatively simple network.

● **What other parameters, if any, affect performance?**

Clearly regularization also affects performance. Learning rate, weight initialization values, lambda (in the case of regularization), batch sizes, epochs trained, are all other hyperparameters that affect performance.

Part 5: Evaluate your network on the final set.

6) To provide an objective measure of your neural network's ability to classify binding sites, a test dataset has been provided (rap1-lieb-test.txt). There are no class labels on these sequences. Your goal is to maximize the separation in scores for the true Rap1 binding sites as compared with the non-sites.

a. Select a final model (encoding, architecture, training regime). This can be the same as your model in Part 3, Part 4, or something completely different.

I decided to use the same model as I've described above (single hidden layer of 8 nodes, ReLu activation in the hidden layer), with L2 regularization and loss function of MSE. I tested on all of the training data using the same quasi mini-batch scheme (using all of the positive data, and mini-batching the negative data) that I've been using. The 17 base-pair sequences for both positive and negative cases are one-hot encoded. Because I am not subsetting the negative data like I was for 5-fold CV, I will train for more epochs so that the network sees more of the total negative data. Specifically, I will train for 500 epochs.

b. Train your final model on the entire training dataset. Run the trained system on the test dataset. For each sequence, output the sequence and its output value from the network, separated by a tab, as follows:

ACATCCGTGCACCATTT 0.927
AAAAAAACGCAACTAAT 0.123

Results saved in file “output.txt”, in the data directory. Pushed to github repo as well.