Neural distance embeddings for biological sequences

E4040.2021Fall.BIOM.report.rr3415.pc2946.hs3164
Rohith Ravindranath rr3415, Philippe Chlenski pc2946, Hannah Steele hs3164
Columbia University

Abstract

The goal of this project was to reimplement several key aspects of a recent paper by Corso et al. concerning a new method for learning embeddings for biological sequence data, as well as to extend the analysis to novel data and train-test split methods. We were able to reimplement and generalize the authors' code, found good sets of hyperparameters, and did preliminary explorations of the aforementioned topics. However, not every aspect of our results agreed with the original paper; in particular, our hyperbolic distance function-based models behaved differently.

1. Introduction

The recent paper 'Neural distance embeddings for biological sequences' by Corso et al. [2] introduces NeuroSEED, a promising new technique for learning representations of biological sequences. The authors boast fast and accurate heuristics for inference on phylogenetically-related sequences such as edit distance approximation and hierarchical clustering.

The main idea of NeuroSEED is to create a Siamese neural network which encodes any two sequences in a dataset so that their distance in embedding space (according to some distance metric chosen by the user) is as close as possible to the true distance (obtained by e.g. computing edit distances directly).

While promising, the authors' results were quite preliminary. Two limitations in particular stand out in their analysis.

First, although they tested their method on three datasets, all datasets were produced from different regions from the 16S rRNA gene, which is known to have very strong structural constraints. Since the value of these learned representations depends on the strength of the underlying manifold (unsurprisingly, this approach fails to compress randomly generated strings of DNA), the stronger-than-normal constraints on 16S rRNA sequence may translate into non-generalizable results. Thus, one of the goals of this project was to investigate the ability of this approach to generalize to other sequences. We successfully ran the NeuroSEED algorithm on a set of 7,010 sequences of the *PheS* (phenylalanine-tRNA synthetase, subchain A) gene. Because of computational and time constraints, we did not explore *in silico* mutation trees or noncoding DNA.

The second limitation of the original paper is the authors' approach to test-train independence. Although there is no commonly agreed-upon approach to train-test independence in a phylogenetic context—indeed, the authors' approach of choosing samples randomly without paying special attention to the relationship between samples is common—it becomes difficult to disentangle desirable forms of overfitting (manifold learning) from undesirable overfitting (e.g. by memorizing distances between pairs of neighbors in the training set). The original datasets used by the authors are densely sampled, and thus it is quite likely that, for any pair of sequences in the test set, the training set contains pairs of very closely related sequences with known edit distance.

To explore this, we introduced a "phylogenetic" approach to train-test split, where entire subtrees of height n are added to the test set. This ensures that the model cannot memorize relationships between closely related sequences, and instead must rely on deeper underlying structure in the data. We expected that the more the model tended to memorize specific distances, the worse the gap between training and test loss would be as n increased. We ran this on the same set of 7,010 *PheS* sequences for values of n from 1 (random) to 8 (test set consists of subtrees of ~256 genomes). Because of memory and time constraints, we did not repeat this analysis on the 16S rRNA dataset.

Additionally, we experimented with different model architectures, hyperparameters and distance functions to determine the most effective way to embed sequences quickly and at scale.

In terms of model architectures, we attempted to replicate three types of embedding models mentioned in the paper, in order to determine which architecture performed best: (i) the linear embedding model, which was made up of one fully connected layer without any non-linear activation function, (ii) the MLP model, which was made up of two fully connected hidden layers with non-linear activation functions and one fully connected output layer, and (iii) the CNN model, which was made up of two one-dimensional convolutional layers with batch normalization and average pooling after each. Within each model architecture studied, we included a dropout layer immediately following the input layer.

For each model architecture, we also performed hyperparameter tuning across the following model hyperparameters: (i) dropout rate, which was tested across all model architectures; (ii) choice of activation function, which only applied to the MLP and CNN architectures; (iii) number of units in the hidden layers of the MLP; and (iv) number of filters used in the convolutional layers of the CNN model, as well as various training hyperparmeters, such as batch size and learning rate, in order to evaluate which hyperparameter selections led to the greatest efficiency, in terms of both performance–i.e. loss minimization–and training speeds.

We also evaluated all of the distance functions studied in the paper for each of the model architectures described above. This essentially amounted to an investigation of different loss functions, as we defined mean squared error using the selected distance metric as our loss.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The ensuing discussion will only focus on the edit-distance approximation component of the Corso *et al.* paper. Other tasks such as hierarchical clustering and consensus string retrieval can be accomplished using very similar computational approaches, so we focused on edit distance as a "canonical" problem for neural distance approximation.

The NeuroSEED approach in general can be characterized as consisting of four user-selected variables united by a common framework:

1. An encoder model
2. A decoder model
3. A distance metric
4. A task-appropriate loss function

Note that, for edit distance approximation, no decoder model is needed and the loss function is fixed to be mean squared error (or some closely related metric, such as root mean squared error) between the true edit distances and the distances between sequences in embedding space.

In the original paper, the test-train split is carried out randomly (i.e. on the leaves of the tree or with *n*=0), and the edit distance approximation is carried out with respect to three genomic datasets, plus one fully random synthetic dataset, using a number of neural architectures and distance functions. The architectures tested in the original paper are:

- Linear (single-layer perceptron)
- Multilayer dense neural network (MLP)
- Convolutional neural network (CNN)
- Gated recurrent unit (GRU)
- Transformer

In addition to these neural architectures, the authors also tested k-mer embeddings, which can be computed directly from the data and have no trainable parameters. For each architecture, the distance functions tested were:

Paper equation 1. The distance functions used in the Corso *et al* paper and their equations.

$$\text{Manhattan} \quad d(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=0}^{k} |p_i - q_i|$$

$$\text{Euclidean} \quad d(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_2 = \sqrt{\sum_{i=0}^{k} (p_i - q_i)^2}$$

$$\text{square} \quad d(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_2^2 = \sum_{i=0}^{k} (p_i - q_i)^2$$

$$\text{cosine} \quad d(\mathbf{p},\mathbf{q}) = 1 - \frac{\mathbf{p}\cdot\mathbf{q}}{\|\mathbf{p}\|\|\mathbf{q}\|} = 1 - \frac{\sum_{i=0}^{k} p_i q_i}{\sqrt{\sum_{i=0}^{k} p_i^2}\sqrt{\sum_{i=0}^{k} q_i^2}}$$

$$\text{hyperbolic} \quad d(\mathbf{p},\mathbf{q}) = \text{arcosh}\left(1 + 2\frac{\|\mathbf{p}-\mathbf{q}\|^2}{(1-\|\mathbf{p}\|^2)(1-\|\mathbf{q}\|^2)}\right)$$

2.2 Key Results of the Original Paper

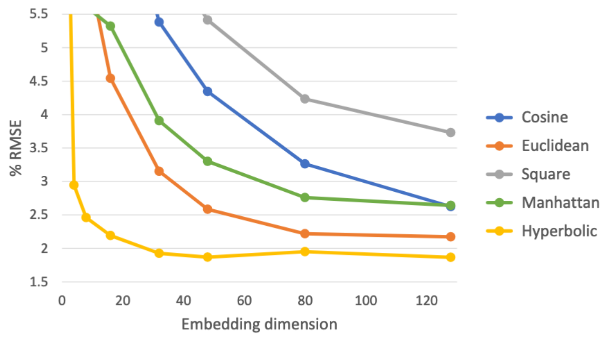
For the genomic datasets, the top three best-performing combinations of architecture and distance function were:

1. RT988 (short genes): global transformer + Manhattan distance; local transformer + hyperbolic distance; global transformer + hyperbolic distance.
2. QIITA (medium genes): CNN + hyperbolic distance; global transformer + hyperbolic distance; local transformer + hyperbolic distance
3. Greengenes (long genes): CNN + hyperbolic distance; GRU + hyperbolic distance; CNN + euclidean distance.

For the synthetic dataset, all methods performed substantially worse; according to the authors, this is because there is no underlying manifold from which the data is being sampled; thus the dimensionality of the manifold in which the random data lies is closer to the observed dimensionality of the data; and thus the inputs cannot be compressed as much.

The authors highlight the importance and novelty of the hyperbolic distance function. On average across all trials, using a hyperbolic distance function offered a 22% reduction in the RMSE compared to the next-best geometry with equivalent hyperparameters. In addition, hyperbolic geometries are "saturated" at lower dimensions than competing geometries, indicating that smaller embeddings are possible under hyperbolic geometries (see Figure 1). In general, this is consistent with observations that hyperbolic geometries are more efficient for embedding tree- and graph-structured data in low dimensions.

Paper figure 1: RMSE reduction as a function of embedding dimension for different distances



(Source: Corso *et al*)

3. Methodology (of the Students' Project)

3.1. Objectives and Technical Challenges

The reimplementation of this project had four main components:

1. Reproduction of some of the main results from the paper (e.g. comparison of distance functions)
2. Hyperparameter search for some candidate architectures
3. Application of NeuroSEED methods to non-RNA data
4. Testing the effect of different train-test splits on the accuracy of the NeuroSEED embeddings

To reduce the computational demands of this project, we limited ourselves to the QIITA datasets for parts (1) and (2) and only tested linear, MLP, and CNN architectures. Since the comparison of distance functions was a central investigation in the original paper, we reimplemented each distance function for an initial comparison, but only used Euclidean or hyperbolic distances for parts (2), (3), and (4).

The main technical challenges anticipated in this project were reimplementing the authors' code using a more straightforward, Keras- and TensorFlow-based approach (the original paper is written in Torch) and adapting their methods to new data/train-test splits. Since the new data could be larger and we presumably had smaller compute resources at our disposal than the authors did, we understood that unanticipated challenges could also arise in terms of memory usage, model complexity, and computation time.

For instance, computing edit distances directly turned out to be too slow (it would have taken on the order of 72 hours per dataset), leading us to re-implement some of the authors' optimizations (multithreading, aggressive vectorization, and the use of the C-optimized Levenshtein package). Many other such compromises are described in subsequent sections.

3.2. Problem Formulation and Design Description

The goal of this project was to reimplement a subset of interesting experiments from the Corso *et al* paper. To achieve this, we have redesigned the neural network as a highly modular Siamese network in TensorFlow/Keras. Specific details about the implementation, including pseudocode and flowcharts, can be found in Section 4.2 of this paper.

4. Implementation

In this section, we discuss the acquisition of data and the deep learning architectures we investigated. The software design section includes an overview of the modular Siamese network, an overview of the preprocessing pipeline, and the details of the data generator we built to mitigate potential memory issues with regard to handling training data.

4.1 Data

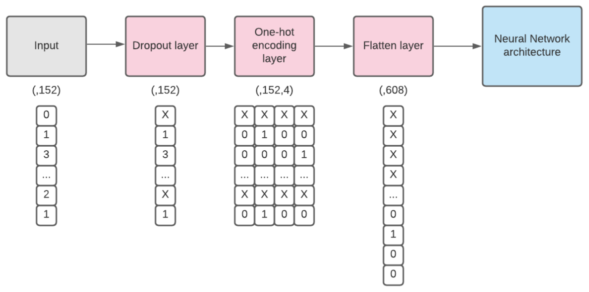
The following datasets were used:

- QIITA dataset, provided by download link from original NeuroSEED github repo [3]
- PheS dataset, generated using P3-scripts to scrape from PATRIC [4]. This dataset contains 7,010 sequences.
- 16S RNA dataset, generated using P3-scripts to scrape from PATRIC [4]. This dataset contains 16,861 sequences. It is roughly equivalent to the GreenGenes dataset used in the original paper.

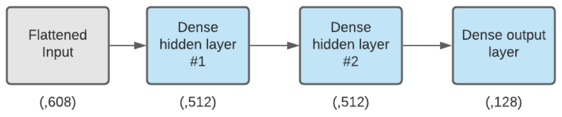
4.1 Deep Learning Network

We evaluated three separate network architectures from the Corso *et al.* paper: a linear model, which does not incorporate any non-linear activation functions, an MLP model, and a CNN model. The first four layers were the same across all three architectures, as they represent transformations of the data that did not require learning any parameters via a neural network layer. The linear model simply added a fully connected dense layer (with no non-linear activation function) of 128 units (to allow for an output embedding of size 128) downstream of the four model-agnostic layers. The MLP network added two hidden layers with non-linear activations, plus a final dense layer to output the embedding vectors of dimension 128. The CNN network consisted of two one-dimensional convolutional layers, with batch normalization and average pooling applied after each convolutional layer, plus a similar final dense layer to output the embeddings of size 128. See flowcharts below for details. Our code to implement these model architectures can be found here on Github, in the get_embedding_model() function.

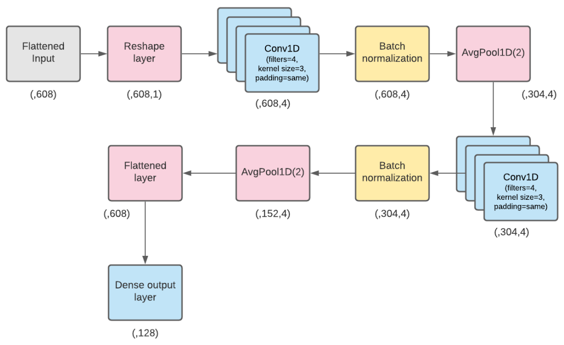
Flowchart 5(a): General embedding encoder model architecture. As an illustrative example here, we show the dimensions of each layer when QIITA data is passed into the network. The QIITA dataset consists of sequences of dimension 152. For the linear architecture, the box labeled "Neural Network architecture" would simply be replaced by one fully connected dense layer, with no non-linear activation function. For the MLP and CNN architectures, this box would be replaced with the architectures in the Flowcharts 5(b) and 5(c), respectively.



Flowchart 5(b): MLP embedding network architecture. Here we show an illustrative example with 512 units in the hidden layers.



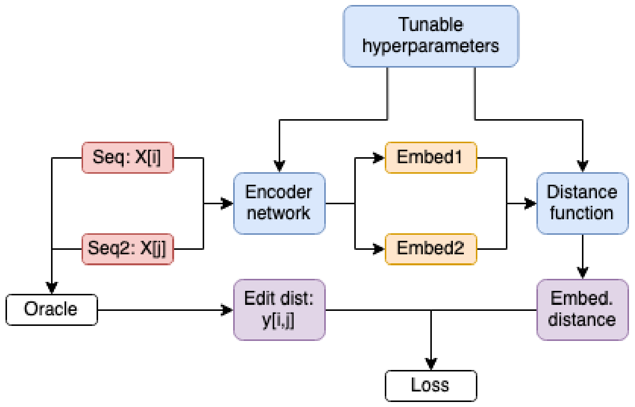
Flowchart 5(c): CNN embedding model architecture. Here we show the network architecture and dimensions of each layer, using 4 filters as an illustrative example.



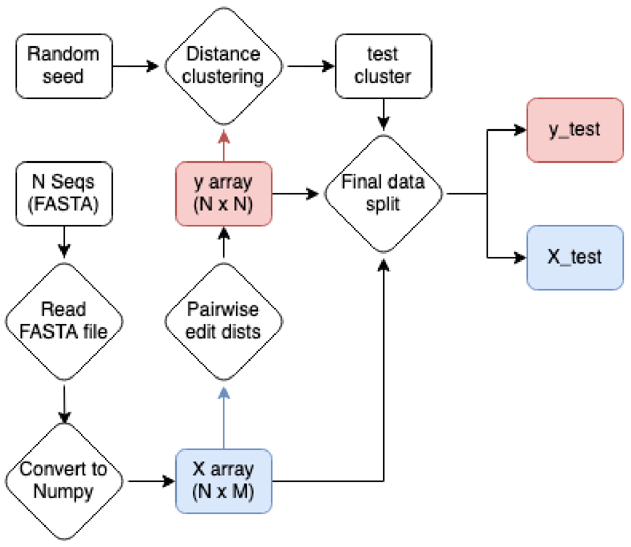
4.2 Software Design

The following flowcharts cover the central aspects of our software design, namely (i) the Siamese network for edit distance approximation; (ii) the data pre-processing pipeline for curating datasets with distance-based splitting; and (iii) the data generator for generating batches of data on the fly during training.

Flowchart 1: Siamese network for edit distance approximation. This is the general architecture we implemented in Keras. The modularized version can be found here on Github as the function train_siamese_model(). The encoder network, distance function, and various training hyperparameters (e.g. the optimizer used to train the model) can all be specified by the user.



Flowchart 2: Preprocessing pipeline (distance-based splitting). One of the key challenges of this project was to preprocess sequence data into appropriate Keras inputs. Converting biological sequences to numpy arrays is relatively common practice; however the distance-based splitting approach is, as far as we know, novel and nontrivial. It can be found here on Github, particularly in the function train_test_split_distance(). The entire pipeline is wrapped in process_seqs(). Please note that, although this flowchart only shows the generation of the X_test and y_test arrays for conciseness, this pipeline actually generates training, test, and validation sets in parallel.

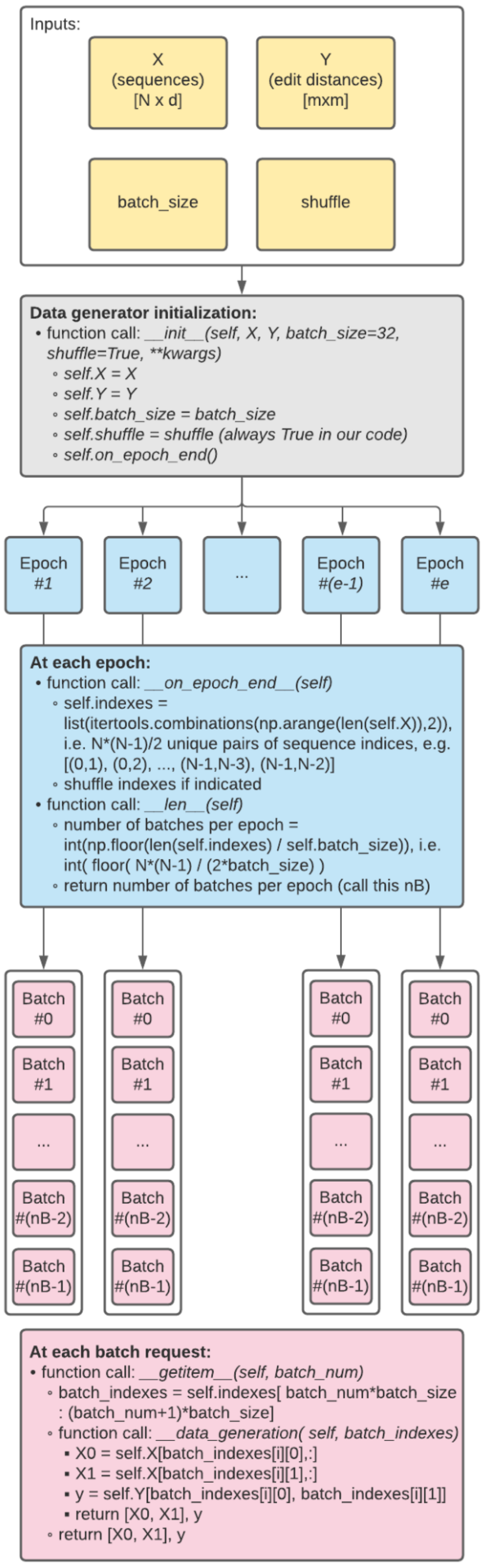


Pseudocode 1: distance-based splitting. These functions break up test and train data such that the ~2ⁿ closest relatives of each test set element also end up in the test set.

```
get_clusters(y, indices, size, depth):
    out = []
    while len(out) < size:
        row = random_row(y)
        n_relatives = 2 ** depth
        top_n = argpartition(row, n_relatives)[0:n_relatives]
        relatives = intersection(indices, top_n)
        out = union(relatives, out)

train_test_split(X, y, test_size, val_size, depth):
    n = len(X)
    indices = range(n)
    n_test = int(test_size * n)
    n_val = int(val_size * n)
    test_indices, nontest_indices = get_clusters(y, indices, n_test, split_depth)
    val_indices, train_indices = get_clusters(y, nontest_indices, n_val, split_depth)
    return X[train_indices], X[test_indices], X[val_indices], y[train_indices], y[test_indices], y[val_indices]
```

Flowchart 3: Data generator workflow. The flowchart demonstrates the main functionality of the data generator, the purpose of which is to generate batches of data on the fly at each training step, which helps to avoid memory issues when working with the large amount of input data required by our model. The Siamese model required unique data preparation steps at each epoch initialization and batch request. Therefore, we consulted a Stanford.edu blog on building data generators, and began from the generic data generator code on that web page [8]. We then modified the class methods to fit the unique needs of our problem. The logic is described in the flowchart below. The code for this data generator class, called SequenceDistDataGenerator(), can be found here.



Data generator initialization:
- function call: *__init__(self, X, Y, batch_size=32, shuffle=True, **kwargs)*
  - *self.X = X*
  - *self.Y = Y*
  - *self.batch_size = batch_size*
  - *self.shuffle = shuffle (always True in our code)*
  - *self.on_epoch_end()*

At each epoch:
- function call: *__on_epoch_end__(self)*
  - self.indexes = list(itertools.combinations(np.arange(len(self.X)),2)), i.e. N*(N-1)/2 unique pairs of sequence indices, e.g. [(0,1), (0,2), ..., (N-1,N-3), (N-1,N-2)]
  - shuffle indexes if indicated
- function call: *__len__(self)*
  - number of batches per epoch = int(np.floor(len(self.indexes) / self.batch_size)), i.e. int(floor(N*(N-1) / (2*batch_size))
  - return number of batches per epoch (call this nB)

At each batch request:
- function call: *__getitem__(self, batch_num)*
  - batch_indexes = self.indexes[batch_num*batch_size : (batch_num+1)*batch_size]
  - function call: *__data_generation(self, batch_indexes)*
    - X0 = self.X[batch_indexes[i][0],:]
    - X1 = self.X[batch_indexes[i][1],:]
    - y = self.Y[batch_indexes[i][0], batch_indexes[i][1]]
    - return [X0, X1], y
  - return [X0, X1], y

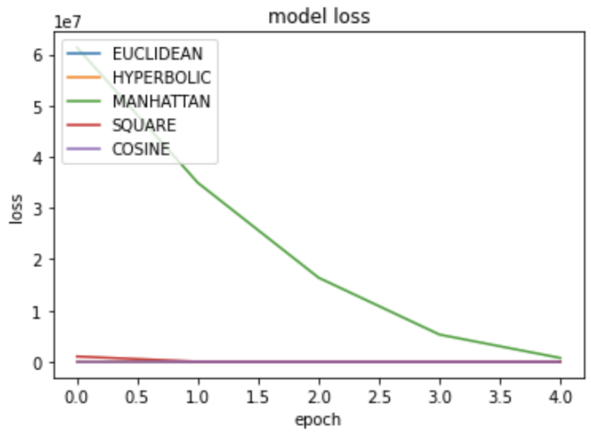
5. Results

5.1 Project Results

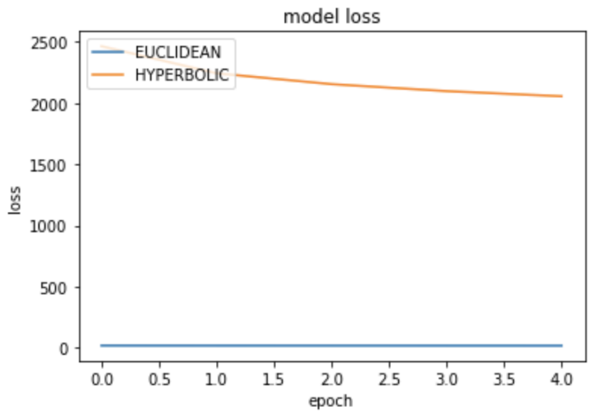
The two main experiments we conducted across our different model architectures were (i) a comparison of distance function metrics, and (ii) hyperparameter tuning of model and training hyperparameters. We wanted to compare and observe how different distance functions perform using different embedding encoder network architectures. While in our distance function tests, we were mainly concerned with loss minimization, our hyperparameter tuning was focused on loss minimization as well as training time reduction, which was a significant consideration in this project.

5.1.1 Distance Functions Experiment

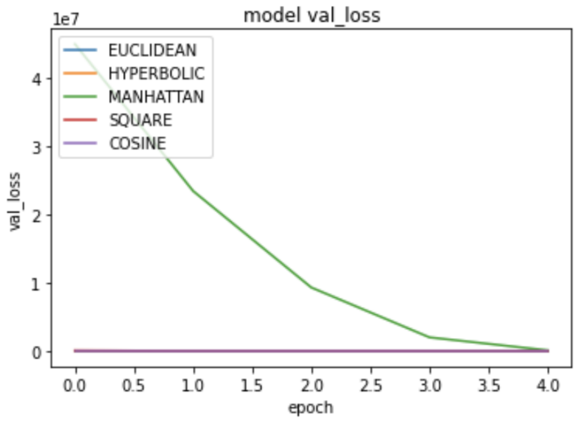
The purpose of our distance function experiments was to see how the model performed under each of the different distance functions studied in the paper. This was interesting to see since each distance made various assumptions about the data. Some functions, such as hyperbolic distance, focused on projecting the data in a multi-dimensional hyperbolic plane. Below are some of the plots of our results with respect to the linear model architecture. We also investigated this across the other two model architectures, but we only present the results of the linear architecture here to avoid redundancy, as the other network architectures demonstrated very similar results.



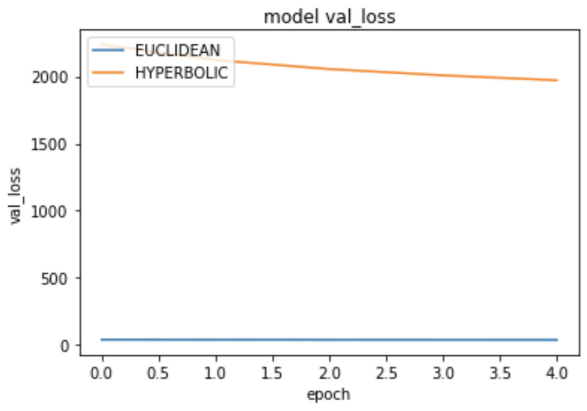
Result Plot 1: Training loss using various distance metrics, for the linear model architecture. (Note: results for the other model architectures can be found here: [linear][mlp][cnn])



Result Plot 2: Training loss using Euclidean and hyperbolic metrics, for the linear model architecture. (Note: results for the other model architectures can be found here: [linear][mlp][cnn])



Result Plot 3: Validation loss using various distance metrics, for the linear model architecture. (Note: results for the other model architectures can be found here: [linear][mlp][cnn])



Result Plot 4: Validation loss using Euclidean and hyperbolic metrics, for the linear model architecture. (Note: results for the other model architectures can be found here: [linear][mlp][cnn])

Plot 1 displays the training loss for all distance metrics; we notice that Manhattan distance is much worse than the others in these plots. Plot 2 shows us a closer view of our two best distance functions, Euclidean and hyperbolic. Notice that the Euclidean metric performs far better than the hyperbolic. Plot 3 and Plot 4 observe the same results and observations as with Plot 1 and Plot 2, but this time with val_loss. This is expected, as one would expect the val_loss to be correlated with the model of the loss during training.

5.1.2 Hyperparameter Search Experiment

	Linear Model	MLP Model	CNN Model
Hyperparameter Tuning with Hyperbolic Distance	act_func: tanh dropout: 0.7 learning_rate: 0.01 batch_size: 512 Score (loss): 977.13	act_func: relu dropout: 0.5 mlp_num_units_hidden: 128 learning_rate: 0.01 batch_size: 256 Score (loss): 54.456	act_func: tanh dropout: 0.7 mlp_num_units_hidden: 512 cnn_num_filters: 3 learning_rate: 0.001 batch_size: 256 Score (loss): nan
Hyperparameter Tuning with Euclidean Distance	act_func: relu dropout: 0.5 learning_rate: 0.0001 batch_size: 512 Score (loss): 35.363	act_func: relu dropout: 0.3 mlp_num_units_hidden: 128 learning_rate: 0.0001 batch_size: 256 Score (loss): 32.989	act_func: relu dropout: 0.3 mlp_num_units_hidden: 512 cnn_num_filters: 4 learning_rate: 0.0001 batch_size: 512 Score (loss): 39.588260650634766

Result Plot 5: Best hyperparameters for each model with respect to hyperbolic and euclidean [linear][mlp][cnn].

For this experiment we ran the RandomSearch method from the keras-tuner library. The hyperparameters we have chosen differ across the different models, as some necessitated additional hyperparameters. The experiment was done over 6 trials, each with a random set of hyperparameters. We considered not to do GridSearch as our search space is massive and given the enormous training time for one model, we wouldn't have received our results on time. One interesting thing to note is that the score for the "Hyperparameter Tuning with Hyperbolic Distance" with respect to the Linear Model is much higher compared to the other models. Notice that in all models, the Euclidean distance still outperforms the hyperbolic distance regardless of the model. We also notice that when using the hyperbolic distance, the learning rate is much higher compared to when using euclidean distance.

5.1.3 Extension to non-RNA data

We successfully were able to run the NeuroSEED pipeline on non-RNA data; specifically, we extended the analysis to the PheS gene.

The following section on train-test splits was carried out on PheS data, and goes into more details about the results we attained on this dataset. Overall, these results are consistent with the results achieved for 16S rRNA, suggesting that this approach does generalize to other coding sequences. Moreover, this demonstrates the effectiveness of the preprocessing pipeline we developed, enabling us to continue using our preprocessing and training scripts with minimal further effort to investigate the properties of other types of sequences. More sequences were not tested during this project because of the prohibitive time and computational costs of computing the ground-truth edit distance matrices and training good embedding models.

Accuracies across datasets are not readily comparable, as the edit distances themselves tend to differ. In order to understand whether NeuroSEED performs "better" or "worse" for different sequences, more work needs to be done concerning the choice of embedding size and rescaling of loss values.

5.1.4 Train-test split experiment

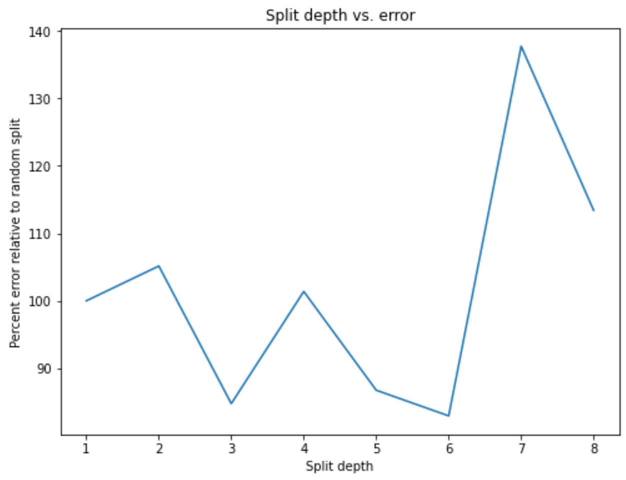
For this experiment, we trained *de novo* embedding models for split depths between 1 (sampling singleton clusters into test and validation sets) and 8 (cluster size of 256), evaluating the test, train, and validation loss values that resulted compared to the baseline value. Each model was trained for 2 epochs using the following settings:

- Dataset: PATRIC PheS dataset
- Distance function: hyperbolic
- Model architecture: linear
- Input dimension: 1821
- Embedding size: 911 (half of input)
- Optimizer: Adam with clipped norms at 1 and learning rate of 1

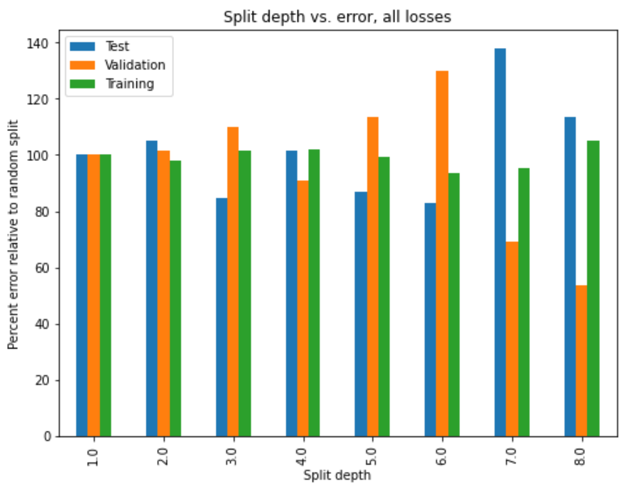
These values were chosen to imitate the results in the tutorial notebook for the original NeuroSEED paper (and especially so that the hyperbolic distance function could be used), with an emphasis on run speed so that several embedding models could be trained successively without running into time constraints.

While the initial relationship was unclear from observing test losses alone, with model performance even appearing to improve as larger clades (depths 2-6) were withheld, a more detailed analysis including the validation loss revealed that this was an artifact of major increases in variance. One reason why the variance could be increasing, rather than the model performance dropping across the board as expected, is that these models have not yet converged over two epochs and thus the performance is more of an artifact of the random seed than of the actual manifold learning process. In the future, more epochs, experiments with smaller datasets (e.g. QIITA), and other distance functions may help reinforce this pattern or bring out new relationships.

Results plot 6. Test error as a function of split depth. Here we see that test error appears surprisingly robust until a split depth of about 7 (126 genomes per cluster), at which point it shoots up. Still, there is a fair amount of noise in this result and it is hard to confidently draw any conclusions from such an experiment.



Results plot 7. All errors as a function of split depth. This plot is normalized so that all errors show as 100% for depth=1. Here we see what's really going on: the variance between train/test/validation errors becomes very large as split depths get deeper; this can be observed at relatively low split depths. Any reduction in split depth observed in the previous plot is explained away by this increase in variance.



5.2 Comparison of the Results Between the Original Paper and Students' Project

In this section, we compare the results of our experiments to the original papers. Specifically, we look at training time and accuracy (%RMSE).

5.2.1 Comparison of Training Time Across Model

	Linear	**MLP**	**CNN**
Our Training Time	38.5 minutes	42.5 minutes	59.5 minutes
Original Paper Training Time	66 minutes	54 minutes	126 minutes

Table 7: Comparison of training times between our models (using Hyperbolic distance) and the original paper's training time [linear][mlp][cnn].

In the original paper when evaluating the models in the edit distance section, they only used hyperbolic distance. As this was the novelty of their paper. Hence when comparing training times, we made sure to only compare our models using the hyperbolic distance. From Table 7, you notice that none of our training times are really matching the original papers. This is because we used different Linear, MLP and CNN architectures in the hopes of potentially improving metrics all around. As you can see, our training times are indeed much smaller than theirs. But what's more important in our findings is that the relationship of the training times between the models is the same in the original paper. The CNN model in our results and paper's results had the longest training time. This makes sense since CNN usually has higher parameters. The Linear and MLP training times are different. This may be because our Linear model has fewer model parameters compared to the one used in the paper. Overall, this comparison showed that across the board we were able to have short training times compared to the paper's model as well as our results showing similar relational characteristics as the paper's results.

5.2.2 Comparison of Accuracy Across Models

	Linear	**MLP**	**CNN**
Our RMSE (converted from SE) Time	20.56518	4.8548	nan
Original Paper Training Time	2.50	1.85	1.56

Table 7: Comparison of paper's %RMSE with our converted %RMSE from SE [linear][mlp][cnn]

$$\%\,\mathrm{RMSE}(\theta, S) = \frac{100}{n}\sqrt{L(\theta, S)} = \frac{100}{n}\sqrt{\sum_{s_1, s_2 \in S} (ED(s_1, s_2) - n\ d(f_\theta(s_1), f_\theta(s_2)))^2}$$

Equation 1: %RMSE formula used in the paper

In the original paper they compared their loss using %RMSE in order to make the numbers comparable over different datasets. However for us, we only used one dataset for our experiments. We used a squared error for simplicity. However, the numbers shown in Table 7 are converted from SE loss to %RMSE. We notice that our losses are higher than the original paper's losses. This may be due to differences in architecture for Linear, MLP, and Dense. We also used different hyperparameters than the ones used in the paper as well, which may have been a contributing factor. We also notice that our CNN model loss is "nan". This is due to numerical stability, which has been solved for the Linear and MLP models, but not for CNN due to time constraints. We also notice a decreasing trend in loss from Linear to MLP to CNN in the paper. This can also be observed in our results too when compared side by side shown in Table 7.

5.3 Discussion of Insights Gained

5.3.1. Input data handling via a data generator

For our edit distance approximation problem, the datasets we are interested in consisted of sequences of base pairs in a particular gene across different organisms. In these datasets, each distinct organism constitutes a sample and each base pair in the genetic sequence represents a feature. In the QIITA dataset, we have 7,000 samples in the training set. Since we are interested in editing distance between different organisms, however, our models require input data consisting of all unique pairs of organisms/sequences. If we have N samples in the original data, then, this means that our input data into the model is N*(N-1)/2 unique pairs. For the QIITA dataset, 7,000 samples turns into 24.4965 million unique pairs of samples. Due to memory considerations, that means that we could not simply pass in all paired training data into the model. Instead, our problem required a data generator to (i) generate the sample pairs from the original samples and (ii) perform the shuffling and batching of those pairs at each epoch, prior to passing in the batch at each training step.

In order to build the data generator, we first consulted a tutorial on building generic data generators to build data batches on the fly at each training step [8]. Using the code in this tutorial as our starting point, we then modified the data generator to:

(i) take in the input data matrix (our X variables) and distance matrix (our y variables) as initialization inputs, rather than a list of IDs used to pull individual data points from separate files, as our data was designed to be loaded in matrix form since the original datasets themselves are fairly tractable to deal with;

(ii) prepare the set of all unique pairs of input data sequences, as this is the full (and much larger) set of data that we need to pass in as inputs to our deep learning model; shuffle over this set of combinations rather than over a set of indices before running each epoch, if shuffling is desired;

(iii) for each batch of data, return X variables that represent a paired list of samples rather than a singular input matrix, and y variables that represent the distance between each pair.

Our data generator was implemented via our customized SequenceDistDataGenerator() class, the code for which can be found here.

5.3.2. Challenges with the hyperbolic distance function

The hyperbolic distance function can be extremely poorly conditioned for various vector inputs. To handle this, the NeuroSEED authors apply boundary conditions on intermediate calculations in their hyperbolic distance calculation and also enforce that each input vector lies within a radius that is itself a tunable model parameter. In our initial hyperbolic distance function specification, we encountered loss that blew up to NaN very quickly. We then added the boundary conditions on intermediate calculations in our function—both the boundary conditions that the NeuroSEED authors utilize in their code as well as additional boundary conditions and approximations when numbers are approaching machine precision limits—and still encountered issues with NaN loss. We then added gradient clipping as an additional protection against NaN loss, which also helped, but we still found that we ran into NaN loss on various occasions.

Eventually, after spending a week debugging the hyperbolic distance function in an effort to prevent the occurrence of NaN loss, we decided that it was best to move on to new areas of the project, as there was no guarantee that we would be able to fix this issue before the project was due, and we needed to ensure that we could generate some results before the project deadline. We suspect that one of the reasons that we may have had more issues with this particular problem than the NeuroSEED authors seemed to have is due to the fact that they implement a layer which enforces that embedding vectors lie within a learnable radius parameter, with another learnable scaling parameter then applied to the final embedded distance to fix the scale difference that the radius enforcement would produce. We suspect that implementing these additional features in our model architectures may have helped to further mitigate this NaN loss issue in training. We also suspect that may have helped bring the hyperbolic distance loss down, as our loss using hyperbolic distance was significantly larger than the loss we achieved using Euclidean distance, which was not the finding of the NeuroSEED authors. However, it seemed like a very large undertaking to explore the authors' PyTorch implementation of these custom layers and then implement them in Tensorflow and Keras, so we chose not to spend time on this particular feature. With more time, however, we would hope that doing so would allow us to replicate the finding of the authors, that hyperbolic distance is the best distance metric to use in terms of minimizing loss in edit distance approximation using embeddings.

5.3.3. Runtime considerations

Early in the project when we tried to replicate the model performance by using the same training settings. We noticed that our training time was over two hours for a single epoch. As expected, this was due to the difference in hardware.

Although we ran our models on the Google Cloud VM attached with a GPA. The paper mentions that they used the GPU (GeForce GTX TITAN Xp). This GPU far exceeds the computational power of our GPU. For this reason, we had to time methods to optimize our training time. One methodology was the data generator. This significantly reduced the training and the amount of the data with the GPU while training,

Another methodology was identifying layers with unneeded too many parameters and substituting with simpler layers. Specifically, replacing the Keras Embedding layer with our own custom one-hot encoding layer. This helped significantly as our custom layer has 0 parameters. This reduced our training time by ~30%.

Lastly we had to reduce our training settings. One specific example is the number of epochs. In the paper's repository, they train the model for 15 epochs. With our hardware and time constraints we could not train the model for 15 epochs, hence we chose only 5. Interestingly, the performance we attained with only 5 epochs was proportionally similar to the paper's result. This may have been due to the slight variations we made to our models.

Overall, we have to make significantly different engineering decisions and changes to our model architecture and training loops in order to meet with the runtime constraints and make sure our experiments and models are completed in a reasonable time.

5.3.4. Special considerations for the CNN network

There were a few special considerations that we needed to deal with in implementing the CNN embedding encoder model. First, we should point out that the CNN we use in this model is a one-dimensional CNN, which is different from the two-dimensional CNNs that we have used throughout this course, but naturally more appropriate for genetic sequences, which, unlike images, are one-dimensional in nature. We selected the CNN architecture for replication in this project (i) it was one of the higher-performing architectures in the paper and (ii) because we suspected that parameter sharing across neighboring features could be beneficial for the encoding model, given that neighboring base pairs in a genetic sequence are translated in sequence, with three base pairs encoding a single amino acid building block. Therefore we used a kernel size of 3 in our CNN model architecture.

The first issue that we encountered in the CNN model development was that we could not simply take in the output of a flattened layer to be used in our first convolutional layer; instead, we needed to reshape to add a dimension (even though the dimension was 1). We were able to fix this bug fairly quickly, but it did take some tinkering and we thought it worth noting that Keras does not automatically handle this for a one-dimensional convolutional layer.

The second, and much more significant, issue that we ran into in our CNN implementation was that the CNN model required far more memory than the other two models. In fact, we could not run the CNN model without upgrading our Google cloud machines to 8 vCPUs and 30 GB of memory. Moreover, the CNN model seemed to be more sensitive to loss blowing up to NaN, even with gradient clipping and bounding in our hyperbolic distance function.

5.3.5. One-hot encoding layer

When developing the embedding layer in our models, we needed to make sure that the embedding one hot encoded the data at some point. This is because our data contains only 0, 1, 2, and 3. Each number identifies with a single DNA nucleotide - A, C, T, and G. However, the model may think of the numbers as some hierarchy and give more weight to one or another, while in fact they have equal weight and do not not have any hierarchy structure between them. This is why one-hot encoding is important to our model. Initially, we used the Keras Embedding layer, as we thought if the vocabulary and out_dim match, then that would result in the Embedding layer one hot encoding the data. However, this was not the case and very apparent we notice the layer having an usually high number or weights for a simple task. Even more so, the model training time was > 1.5 when using the Embedding layer. For this reason and to be more accurate with what the paper was doing, we implemented our own keras Layer, where it does a single operation to one-hot encode the data. Two advantages were seen in this change. One was the runtime was significantly decreased and the model parameters were cut in half. Overall, this was a good decision made by the team. You can see how we implemented the custom one-hot encoding layer here.

5.3.5. Custom Hyperparameter Tuning

When constructing the hyperparameter search test, we had to decide on whether to do a GridSearch or a RandomSearch. GridSearch usually gives the best hyperparameter as it goes through every possible combination in the search grid. However, given that all our models take roughly >45minutes and we have multiple parameters for each model, it would take days for the GridSearch to complete, which is time we don't have. Hence, we decided to go the RandomSearch route. RandomSearch still produces good model settings but it doesn't take much time. We also limited the number of trials to 6 for each search in order to reduce the run time. Furthermore, since we are using keras models, we used a library called keras-tuner [7] to implement our search. We had to implement a custom Tuner class in order to incorporate our data generations and how we build our model, since our model doesn't have a typical keras architecture. We also had a discussion in whether to include cross-validation in our tuner search, but we decided not to due to time constraints. We also had to decide the search space for each of the hyperparameters. One important thing we had to consider was our big dataset. Due to that consideration, we limited the search space for each of the hyperparameters by only subsetting the options that would prove advantageous for a full (and big) dataset. You can see how we implemented our tuner class here.

6. Future Work

This project has been extremely promising for demonstrating the feasibility of applying a NeuroSEED-like approach to other data. Many of these tasks have been wrapped up in convenient utility functions. Certain things that were cut due to time constraints would be top priorities for future work:

- Applications of NeuroSEED to more datasets, including more sophisticated methods for comparing losses across datasets and *a priori* choices of embedding dimension.
- More downsampling analysis for other datasets, architectures, and distance functions.
- Downsampling based on phylogenetic tree rather than distance (code for this is implemented, but we have not found an appropriate dataset. The issue is that few, if any, neural-size datasets have complete phylogenetic trees available).
- More hyperparameter search, including extension of the architecture to transformer/GRU/RNN/LSTM models.

The following future directions are more theoretical and outside the original intended scope of the project. Nonetheless, they appear to be promising research directions:

- Further troubleshooting of the hyperbolic distance function.
- Thorough comparisons between linear encoding vs. taking the top n principal components to create an n-dimensional embedding.
- Applications of novel geometries for shotgun metagenomics or embedding multiple genes/genome regions at once.
- Applications of neural distance embeddings to metagenomic data, e.g. by adding 16S

embeddings together in proportion to their relative abundances.

Finally, we have not touched on a number of research directions identified in the original paper. We underscore them here as inspiration for future researchers:

- Generalization of the edit-distance framework to use different (inexact) oracles in order to speed up the training data generation process.
- The explicit use of hyperbolic graph neural networks to further exploit the geometry of the hyperbolic embedding space.
- The downstream use of NeuroSEED-style embeddings for deep learning tasks on biological sequence data. (It may be possible to use the trained embedding models from our framework directly as a component of a downstream neural net here.)

7. Conclusion

The goal of this project was to reimplement the NeuroSEED architecture outlined in the Corso *et al* paper, turning it into a modular and legible set of Python scripts. We accomplished this in Keras and Tensorflow, implementing all of the functionality of NeuroSEED except for decoder functions. (Any neural network could be used for the encoders; we explicitly coded linear, multilayer dense, and convolutional architectures to test.)

While much of our data and analysis matched the authors, some of the performance of the hyperbolic distance function diverged. Everything else passed our sanity checks and comparisons to the original paper.

In addition to comparing and reimplementing key elements of the original study, we made a number of novel contributions to the study of neural embeddings in biology. In particular, we performed further testing on different NeuroSEED parameters, improving our understanding of what approaches are most effective for generating embeddings. Additionally, we explored the dependence of NeuroSEED on observing closely-related sequences in the training set, helping advance theoretical and mechanistic understanding of the way NeuroSEED embeddings seek the manifolds underlying sequence data. We hope that these contributions will help understand how to generate good embeddings, under what circumstances NeuroSEED-based models can generalize, and in what circumstances researchers may wish to use such embeddings.

6. Acknowledgement

Provide acknowledgements such as support, help, or assistance from online resources, TAs, colleagues.

7. References

[1] Rohith Ravindranath,Philippe Chlenski,Hannah Steele (2021)ECBM4040NeuroseedClassroom[Source Code].]https://github.com/ecbme4040/e4040-2021fall-project-BIOM-rr3415-pc2946-hs3164.
[2] Corso, Gabriele, et al. "Neural Distance Embeddings for Biological Sequences." *Advances in Neural Information Processing Systems* 34 (2021).
[3] Gabriele Corso(2021) Neuroseed [Source Code]. https://github.com/gcorso/neuroseed.
[4] Davis, James, et al. “The PATRIC Bioinformatics Resource Center: expanding data and analysis capabilities.” *Nucleic Acids Research*. 2020
[4] “Python-Levenshtein,” *PyPI*. [Online]. Available: https://pypi.org/project/python-Levenshtein/. [Accessed: 22-Dec-2021].
[5] “ETE3,” *PyPI*. [Online]. Available: https://pypi.org/project/ete3/. [Accessed: 22-Dec-2021].
[6] “Biopython,” *Biopython · Biopython*. [Online]. Available: https://biopython.org/. [Accessed: 22-Dec-2021].
[7] “Introduction to the keras tuner : Tensorflow Core,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/tutorials/keras/keras_tuner. [Accessed: 22-Dec-2021].
[8] A. Amidi and S. Amidi, *A detailed example of how to use data generators with Keras*. [Online]. Available: https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly. [Accessed: 12-Dec-2021].
[9] Rohith Ravindranath,Philippe Chlenski,Hannah Steele (2021)ECBM4040Neuroseed[Source Code]. https://github.com/rohithravin/ECBM4040-NuroSEED-Proj

8. Appendix

8.1 Individual Student Contributions in Fractions

	rr3415	pc2946	hs3164
Last Name	Ravindranath	Chlenski	Steele
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Developed and coded hyperparameter tuning tests and analysis (ran the MLP and Linear notebooks)	Overall conceptualization; most of the non-experiment-specific body of the report (intro,	Data generator code development and report descriptions

		conclusions, etc)	
What I did 2	Developed and coded distance function tests and analysis (ran the MLP and Linear notebooks)	Initial Keras implementation of the Siamese model for NeuroSEED; skeleton code for modularized model.	Code and discussion of embedding model architectures (including searching through NeuroSEED code to understand architectures studied)
What I did 3	Wrote the distance functions as formulated in the original paper (hyperbolic, cosine, euclidean, manhattan, square)	All preprocessing code; all analysis having to do with train-test split variations	Work on improving conditioning for hyperbolic dist function (e.g. tested bounding in the distance function, gradient clipping, etc.) + write-up of this work

8.2 Support Material

While working on this project, we created a public Github repository. This is where we did the bulk of our project work and pushed incremental changes, as one of our group members could not join the classroom repository for some time due to technical issues. If you would like to check this repository, which is public, you will find that the only contributors to the repository are the members of this project, which are the same members (GitHub usernames) in the classroom github repository. Once our codebase and analysis was complete, we moved everything to the GitHub classroom repository. If you would like to view our commits and backtrack our progress, please feel free to do so on our public repository [9].