Protein Residue Distance Matrix Prediction

Andersen Chang (ac7244)
Jaime Abbariao (jma709)
Rajeev Joshi (rj1234)

Abstract

In predicting the distance matrix between residues in a protein, initial baseline algorithms such as K-NN with LCS work to a degree, but can be seemingly improved by leveraging recurrent neural networks. In our case, a many-to-one LSTM proved to be a good baseline model for predicting the average distance between any pairwise nodes on the matrix where the resulting output matrices achieved an RMSE of 3102.17. We do not however want the average data point. We instead want to predict the upper/lower triangular elements of the distance matrix. By considering the primary and secondary sequence as independent inputs and preprocessing them through 1D convolutional blocks we can concatenate them to become a sequence to feed into our LSTM layers. This technique gave us an RMSE of around 2833 with a sensible distance matrix.

1. Introduction

We are tasked with predicting the distance matrix between residues of a protein structure given the protein's primary and secondary structures.

For training, we are given 4554 data rows which consists of two equal length sequences of characters. The first sentence of characters has some variable length n_i that span over a set of 21 letters. This first sentence represents the protein's primary structure, the amino acid sequence. The second sentence of characters has the same length n_i but spans over a set of 8 letters. The second sentence represents the protein's secondary structure. We are also given that for each row i, there is a corresponding matrix M_i which contains pairwise distances between each node in the protein structure which hence gives it the dimensions $n_i \times n_i$.

For testing, we are given 224 data rows to test with the primary and secondary sequences. These sequences again are of equal length.

However, we should take note that sequence lengths can differ which then results in matrices with different dimensions. This is why we denote length with subscript i.

Given that we were working with text data where we were considering similarity, we gave considerations to algorithms such as K-Nearest Neighbors. Given that changes between sequences were by the edit, we thought of leveraging distance metrics like hamming distance. However, other similarity algorithms such as LCS (longest common subsequence) should be considered.

Since we were given K-NN as a baseline approach, we also wanted to try approaches that were newer to us so we had neural networks in mind. We were given text data, so we thought that recurrent neural networks would work in this case as we were interested in the carry-over of state as we wanted to remember the processing of input from previous time steps. However, to deal with the vanishing gradient problem, we employ the use of LSTM and GRU layers instead of a standard RNN.

Given that the training time we have is hampered due to the long sequences, we should also consider how to perform optimizations with our recurrent neural network approaches. This is why we also introduce methods using convolutional layers to read long sequences condense them into shorter sequences as a pre-processing step.

2. Model Results

We experimented with K Nearest Neighbor and RNNs with LSTM to predict the distance matrices given the protein sequence and their secondary structures. The idea behind these baseline approaches is that we want to take an averaging approach to predicting the matrix. That is, we want to find the possible average values that the elements of the matrix would undertake.

However, we broke away from the average-based prediction baseline. In our subsequent models, we wanted to predict the upper or lower triangular elements of the distance matrix. We were allowed to make this assumption simply because we had a symmetric matrix with 0s on the diagonal. We decided to use the LSTM and GRU models to further our experimentation. To capture more information from scanning the sequences, we also decided to use 1D convolutional blocks. Our final model aggregates most of these techniques such that we combined large and small 1D convolutional blocks with Bidirectional LSTM layers and a

fianl output layer to get the lower/upper triangular elements of our distance matrix.

2.1. K-Nearest Neighbours

In using K-Nearest Neighbors, an important metric would be how we are defining distance. In the baseline approach provided to us, K-Nearest Neighbors was implemented with finding the LCS (longest common subsequence). This means that given some sequence in our test set, we want to look for the length of the LCS within the training sequence. Given those lengths, we want to take the k longest ones as we want to obtain the matrices for which the training sequence and test sequence are more similar than others. We then average the k matrices for that one test sequence. This matrix is then later reshaped to our desired dimension.

Another approach we may try is to define another distance metric such as hamming distance and see how that works compared to the longest common subsequence.

2.2. LSTM

We used the LSTM with a many-to-one architecture by taking averages of the given train matrices and and training our network on the scaled and combined (Sequence and q8) sequences in the train data set against their matrix averages.

2.2.1 LSTM Baseline

In this approach we mapped all of the characters in the sequence to their ascii values and then merged all the input row sequences with the q8 sequence and then scaled each column value to the maximum length sequence in the data set, these scaled and combined sequences act as our training inputs for the LSTM layers. For the training output we read all the matrices and made an array of average values in each matrix(corresponding to the sequence). Then we reshaped the input parameters suitable for feeding to the LSTM layers and built a sequential model to predict the outputs for a given sequence(combined protein sequence + q8). Now the model predicts a single output value for a given sequence which can be converted to a matrix with diagonals set to zero.

This baseline approach gives us an RMSE of 3102.17

2.2.2 Stacked LSTM Model

This baseline approach however is not ideal. We are not taking advantage of the features given in the primary sequence or the q8 (secondary) sequence. In our own implementation of the model, we opt to use an alternative tokenization technique from the baseline LSTM model. We rely on using Keras's Tokenizer object and also our own bi-gram function.

Given our primary and secondary sequence, we can construct a bigram from both the primary and seconday sequence and concatenate them to preserve structural information. Once we have this list of bi-grams, we then feed these texts into our tokenizer object for fitting. Once we have the tokenizer fitted, we would then transform the list of texts into integer sequences and pad them according to the longest possible sequence in the training set.

After the preprocessing stage, we set-up our model which was written in Keras. We have the following table to describe our model:

Layer (type)	Output Shape	Param#
LSTM	(None, 1, 128)	419840
Activation('elu')	(None, 1, 128)	0
LSTM	(None, 1, 128)	131584
Activation('elu')	(None, 128)	0
Dense(1, 'relu')	(None, 1)	129

To test this model, we've trained the model on 4300 input rows and have left the remaining 224 for testing the model. Given the predicted average, we again constructed a matrix with 0 on the diagonals and filled every other value with the predicted value. This average-based matrix gives us great results as upon testing the model on the 224 rows, we have achieved an RMSE of 2806.46

2.2.3 Bidirectional LSTM Model

Given the same tokenization process in our latest model, we wanted to further explore the probable use of Bidirectional LSTM. We were thinking that analyzing the sequence in both directions could possibly bring some improvement as we can learn something from the entire context of the sequences. The architecture will still be based on the prediction on an average value for the matrix, and will be similar in form to the first Stacked LSTM model.

Layer (type)	Output Shape	Param#
Bidirectional(LSTM)	(None, 1, 128)	419840
Activation('elu')	(None, 1, 128)	0
Bidirectional(LSTM)	(None, 1, 128)	131584
Activation('elu')	(None, 128)	0
Dense(1, 'relu')	(None, 1)	129

Upon fitting this model and predicting an average matrix value, we find that the model achieves an RMSE of 2931.42.

2.2.4 Many-to-Many LSTM

To further expand on this model, we didn't want to just rely on predicting the average, but instead we wanted to predict a vector of values. In this case, we thought of predicting the elements of the upper triangular matrix (not including the

zero diagonal). We know that the distance map is a symmetric matrix with 0s on the diagonal. So to leverage this information, we instead just predict the upper triangular portion as this reduces the number of predicted values by about half as we know that given a matrix with dimensions $n \times n$, the size of the upper triangular matrix is $\frac{n \times (n-1)}{2}$.

In the preprocessing stage, we use the familiar tokenization technique from the Stacked and Bidirectional LSTM models. We also then do some preprocessing on the training output matrices. For each matrix, we only take the elements of the upper triangular matrix and squash them into a sequence. We then pad these sequences with 0s as needed.

Our model structure will be similar to the previous two, but instead of a Dense layer with 1 output, we then have a TimeDistributed Dense layer with $\frac{(\text{max sequence length}) * (\text{max sequence length} - 1)}{2}$ hidden neurons to serve as our output.

Layer (type)	Output Shape	Param#
Bidirectional(LSTM)	(None, 1, 691)	3822612
Activation('elu')	(None, 1, 691)	0
Bidirectional(LSTM)	(None, 1, 691)	3822612
Activation('elu')	(None, 691)	0
TimeDistributed(Dense))	(None, 238395)	164969340

We can then construct a symmetric distance matrix with dimensions $\frac{(\text{max sequence length}) * (\text{max sequence length} - 1)}{2} \times \frac{(\text{max sequence length}) * (\text{max sequence length} - 1)}{2}$.

However, when we run through the test set, we know that the dimensions of the matrices in the test set may be smaller or larger than those in the training set. We can do two things here:

1. If the dimensions of the test input are smaller, then we can crop the matrix.

2. If the dimensions of the test input are larger, then we can extend the size of the output vector by padding the end values with the average of the output vector.

Our initial implementation of this model came back with worse results than the previous two models coming in with an RMSE of 3624.40.

2.3. GRU

GRU (gated recurrent units) are an alternative to LSTM (long short-term memory) and from what was discussed from class basically do the same thing. However, we have that GRUs are easier to train that LSTM because there are less parameters that we need to tune as the "forget" and "input" gates are merged into a single "update" gate.

We were experiencing training times of about 1-3 hours depending on the type of LSTM model. Therefore, GRU would seem like a good choice so we wouldn't have to spend too long training the model.

2.3.1 Replacing Many-to-Many LSTM with GRU

As a refresher, our previous many-to-many LSTM model took the primary and secondary structures augmented together as the input and went through a series of stacked LSTM layers with activation layers in-between. We would then output a vector of size $n \cdot (n-1)/2$ where n will be the length of the longest possible sequence. To use the GRU layers, we will simply replace all the LSTM layers with GRU.

Layer (type)	Output Shape	Param
Bidirectional(GRU)	(None, 1, 691)	3822612
Activation('elu')	(None, 1, 691)	0
Bidirectional(GRU)	(None, 1, 691)	3822612
Activation('elu')	(None, 691)	0
TimeDistributed(Dense))	(None, 238395)	164969340

The model training was significantly faster than that of the model with LSTM layers. We went from 3 hours of training time to around 30 minutes. This wasn't the only improvement we saw. Given the GRU model, we also saw significant improvement in performance. The previous LSTM model gave us an RMSE of 3624 and using the GRU layers helped us go down to around 3010.

2.4. Multi-Input Models

For the previous models, we've been preprocessing the data and concatenating the two sequences together by an n-gram or by just concatenating them together. However, we believe that we can make some use of the data by creating a model that will take them as separate inputs and perhaps feed them to some preprocessing layers before we do end up concatenating them.

In doing this, we created a helper function called *RNNBLOCK* which would apply the following layers:

Layer (type)
Bidirectional(GRU(128))
Activation('elu')
Bidirectional(GRU(128))
Activation('elu')

In our case we have a GRU layer in there, but we could swap that out for an LSTM if we wanted to, but for the sake of training, we decided to just go with the GRU layer.

2.4.1 Multi-Input GRU and LSTM

We then have the following model using the *RNNBLOCK* function, a concatenation layer, and the output layer.

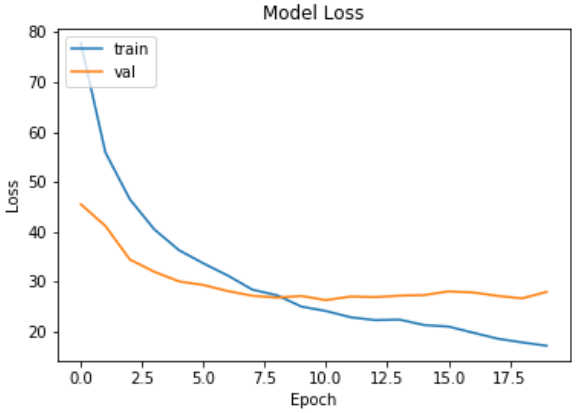

Figure 1. Predicted (top) and Ground Truth (bottom) for Multi-Input GRU Model

Layer (type)	Output Shape	Param
Input()	(None, 1, 691)	0
Input()	(None, 1, 691)	0
RNNBLOCK	(None, 1, 128)	98688
RNNBLOCK	(None, 1, 128)	98688
Concatenate	(None, 1, 256)	0
TimeDistributed(Dense))	(None, 238395)	61267515

So given this approach, we apply the RNNBLOCK to both inputs and then concatenate the results after going through the RNNBLOCK from which we get our desired output vector.

This approach captured more structure in our resulting matrix when tested on a holdout set. However, our RMSE suffered as it skyrocketed to around 3900. But the fact that we did capture more structure shows us that we should be using the primary and secondary structures independently to capture structure more consistently as we need to maximize the information we're getting from our data given that we don't have too much of it.

2.5. 1D Convolutions with RNNs

Given that the sequences we have are still rather large, we want to find a way to capture the temporal patterns within the data and subsequently shorten the sequence. This is where the idea of using a 1D convolutional layer came to mind. RNNs are expensive for processing long sequence, but 1D convnets are cheap. So before we throw the input into the RNN layers, we can first preprocess the sequences using 1D convnets.

2.5.1 1D Convolutions with GRU and LSTM

In our first attempt, we would take the primary and secondary sequences in two separate input layers. We would

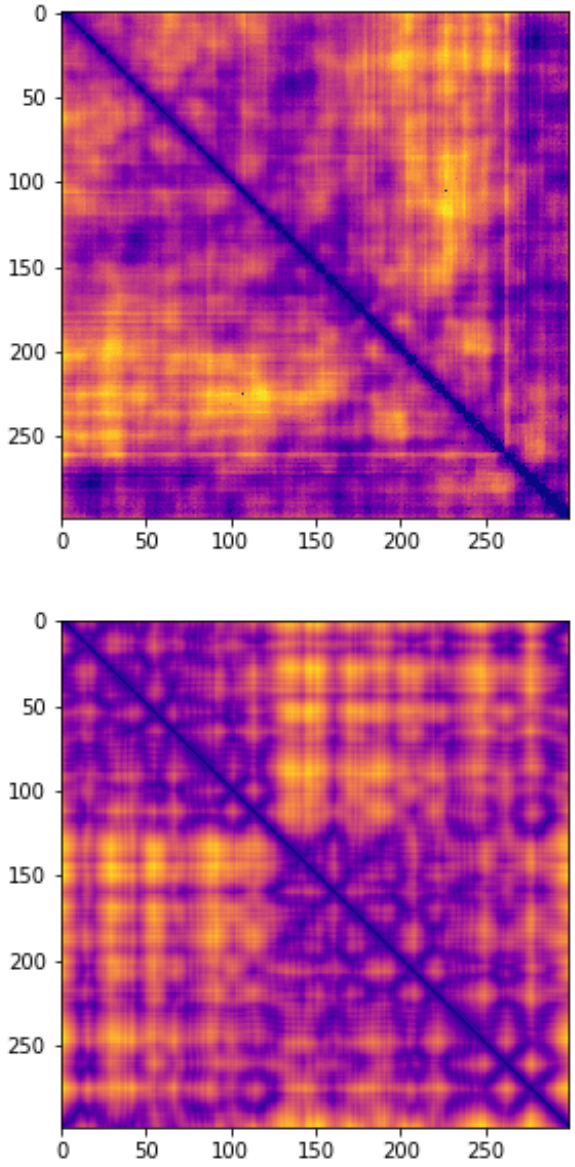

Figure 2. Predicted (top) and Ground Truth (bottom) for Multi-Input GRU Model

then concatenate them such that we have a $2 \times N$ matrix we can put through the 1D convolutional layer. We would then use a MaxPooling layer to get just one row of values. We will then feed a stacked GRU layer these values and then have an output layer that gives us our upper/lower triangular elements.

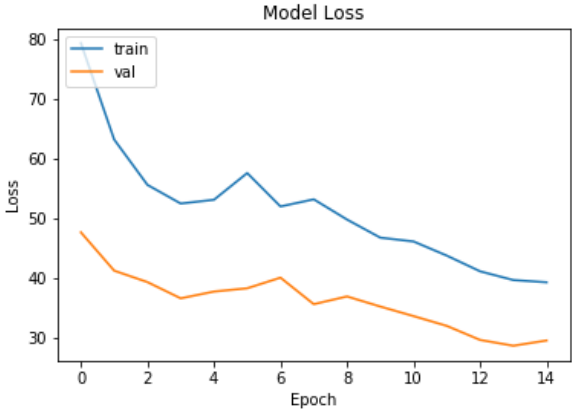

Figure 3. Validation loss and training loss for GRU with 1D Conv + Max Pooling

Layer (type)	Output Shape	Param
Input()	(None, 1, 700)	0
Input()	(None, 1, 700)	0
Concatenate	(None, 2, 700)	0
conv1d	(None, 1, 128)	179328
max pooling1d	(None, 1, 128)	0
GRU	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
GRU	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
GRU	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
TimeDistributed(Dense))	(None, 244650)	31559850

For this model, we find that performance would vary depending on how many convolutional layers we would decide to put in. If we stacked the convolutional layers, we would find better performance compared to just one convolutional layer. For example, with one convolutional layer, we find that we achieve an RMSE of about 3999; however, it's more apparent that we pick up on some structure in the distance matrix. When we decide to increase the number of convolutional layers, we get a lower RMSE of 3650, but we see less of the structure in the distance matrix.

However, this model did give us the idea of preprocessing the primary and secondary sequence individually using 1D convolutional layers before concatenating that result into some input for the RNN layers.

2.5.2 Large 1D Convolutions with LSTM

The final model we created was based on the idea of preprocessing the input sequences through 1D convolutional layers before feeding them into the RNN layers. We started with one convolutional block. But we then thought that we could make use of more convolutional blocks to get a wider array of patterns. So we decided to then implement a large convolution block which would look for temporal patterns with three kernel sizes: 1, 3, and 5. We would then take the three outputs from these convolutional layers and the original text input and concatenate them. We would apply these convolutional blocks to both the primary and secondary sequence. We would then take the output from those blocks and then concatenate the result and feed that into the RNN.

We abstracted the large convolutional block into its own function and those layers are modeled as follows:

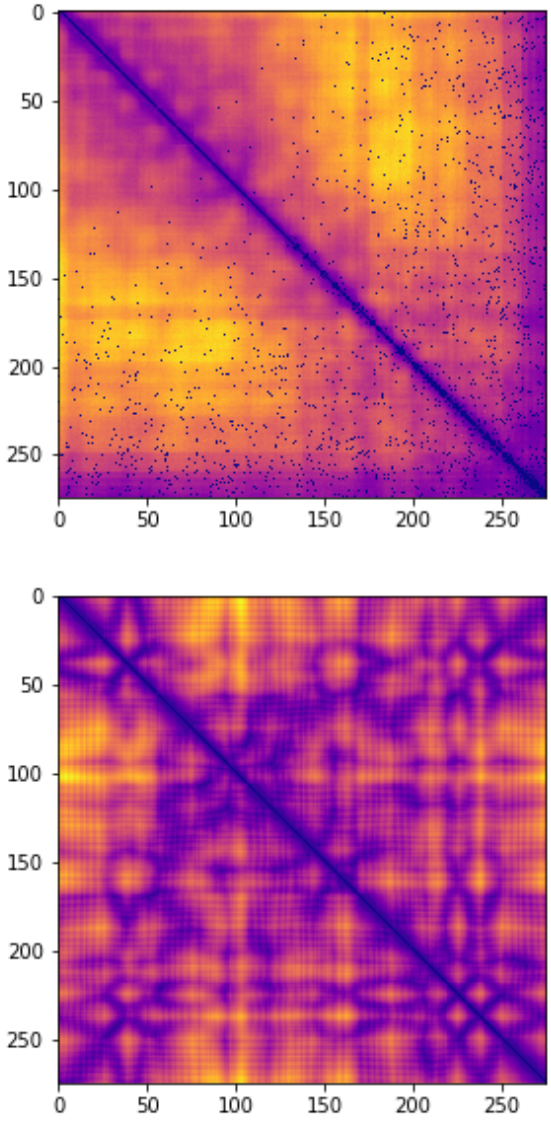

Figure 4. Predicted (top) and Ground Truth (bottom) for GRU with 1D Conv + Max Pooling

Layer (type)	Output Shape
conv1d	(None, 1, 32)
conv1d	(None, 1, 64)
conv1d	(None, 1, 128)
TimeDistributed(Activation('relu'))	(None, 1, 32)
TimeDistributed(Activation('relu'))	(None, 1, 64)
TimeDistributed(Activation('relu'))	(None, 1, 128)
TimeDistributed(BatchNormalization)	(None, 1, 32)
TimeDistributed(BatchNormalization)	(None, 1, 64)
TimeDistributed(BatchNormalization)	(None, 1, 128)
Concatenate	(None, 1, 924)

Notice here that we're concatenating all outputs from the three convolutional layers and the initial sequence which has length 700 which is why we have the output size: (None, 1, 924).

As stated before, we then passed in the primary and secondary sequences individually to these blocks and then concatenate the outputs from these blocks and pass those into the RNN layers. Here is the model as follows:

Layer (type)	Output Shape	Param
Input()	(None, 1, 700)	0
Input()	(None, 1, 700)	0
LargeConv	(None, 1, 924)	0
LargeConv	(None, 1, 924)	0
Concatenate	(None, 1, 1848)	0
Bidirectional(LSTM)	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
Bidirectional(LSTM)	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
TimeDistributed(Dense))	(None, 244650)	31559850

This initial model would give us a pretty solid performance. We would achieve an RMSE of 3000 with this model. The distance matrix would have some figment of structure in them, but it wouldn't be immediately apparent. Given the idea we had before of increasing the number of convolutional layers to get better performance, we decided to do it here as well, but instead we'd be stacking the large convolutional block along with a small convolutional block. We'd then do this several times. We thought of including the small convolutional block after the large convolutional block to capture the patterns caught by the previous block in a more condensed manner.

Our small convolutional block is modeled as follows:

Layer (type)	Output Shape
conv1d	(None, 1, 64)
TimeDistributed(Activation('relu'))	(None, 1, 64)
TimeDistributed(BatchNormalization)	(None, 1, 64)
Concatenate	(None, 1, 988)

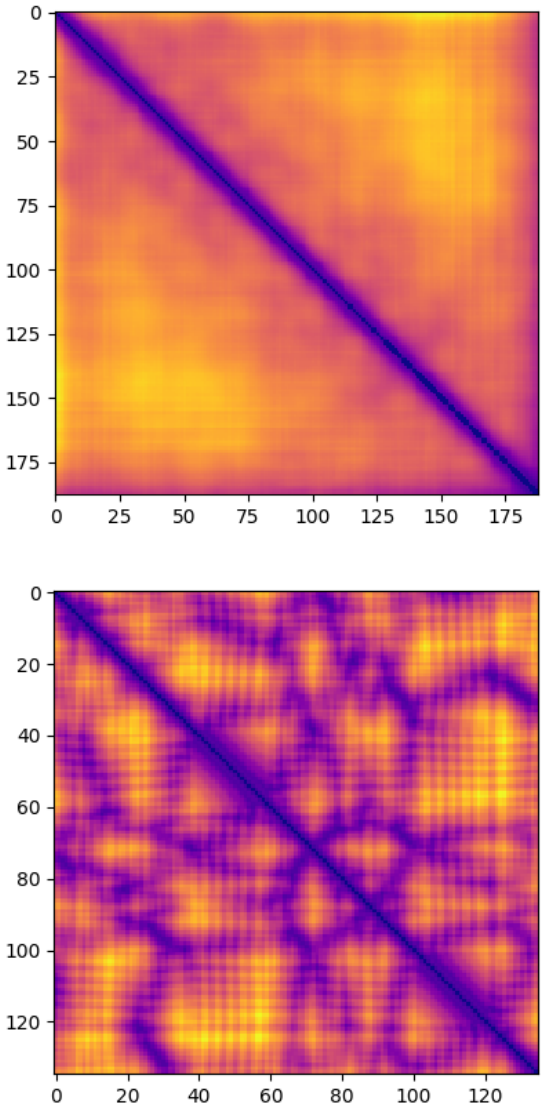

Figure 5. Predicted Matrix on top. Ground Truth on bottom. 1D Convolutional Blocks + LSTM

For both the large and small convolutional block, we add a batch normalization layer to help with speeding up the training. We also add a dropout layers to try to prevent overfitting.

This combination of large convolution and small convolution before passing them into the LSTM layers proved to be worth the experiment as we achieved desirable results with this method.

First of all, here is what the model architecture looked like:

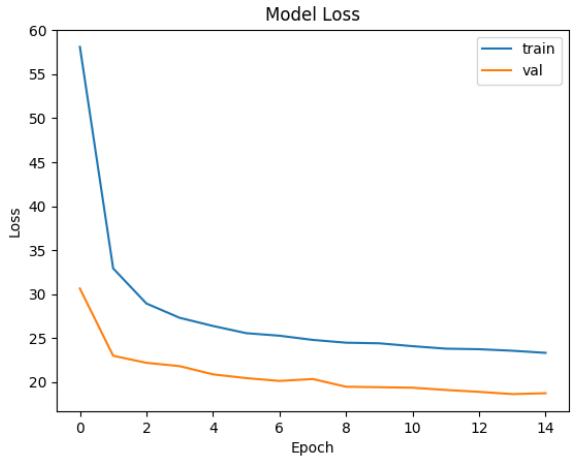

Figure 6. Loss and Validation loss for Large + Small 1D convolutional blocks + LSTM

Layer (type)	Output Shape	Param
Input()	(None, 1, 700)	0
Input()	(None, 1, 700)	0
LargeConv	(None, 1, 924)	0
LargeConv	(None, 1, 924)	0
SmallConv	(None, 1, 988)	0
SmallConv	(None, 1, 988)	0
Concatenate	(None, 1, 1876)	0
Bidirectional(LSTM)	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
Bidirectional(LSTM)	(None, 1, 128)	98688
Activation('relu')	(None, 1, 128)	0
TimeDistributed(Dense))	(None, 244650)	31559850

This model helped us achieve our first sub-3000 RMSE while using separate input layers that are to be preprocessed. On our hold-out set, we achieved RMSE of 2970. The distance matrices that we outputted show better structure than that of the previous model with just the large convolutional block.

To improve on this model, we decided that we should try stacking more of the large conv - small conv blocks together. This ended up working for us as we dropped out RMSE down to around 2833 on our last attempt. The matrices could pick up some sort of structure going on, but it wasn't any different to the previous iteration of the convolution + LSTM model

[2] [1] [3]

References

[1] J. Brownlee. Time series prediction with lstm recurrent neural networks in python with keras, 2016. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/.

[2] F. Chollet. Deep learning with python, 2017.

[3] S. Raschka and V. Mirjalili. Python machine learning, 2017.