

COSC440

dlo54

October 2024

1 Experimental Design and Methods

1.1 Model Structure

The basic structure of the model is that the data is processed into two representations: a pair and an individual representation.

An embedding layer first encodes the individual representation into 18 dimensions before adding positional encoding.

This is then put through 2 transformer encoder layers that do global self-attention and then feed forward.

While the pair representation is generated by adding an embedding of the one hot encoding with a convolution of the individual representation to create a 6-dimensional representation, and this is tiled and added so that if this representation at i is $x[i]$, then the pair representation y at i,j is $y[i,j] = x[i] + x[j]$.

This pair embedding then has a 2D positional encoding added to it, and finally the log of the distances between the things it is encoding appended to create an 8D encoding. The log is done because it appears that distances that are very close to each other have a strong likelihood of being near 0, while things that are just a bit far away in the string lose this very fast.

Both encoding are then put through 3 model layers that each: Put pair encoding through 3 convolution residual layers (don't use attention because you don't have any gpu memory) (at the end of each residual layer, I make sure to enforce that the pair representation transpose is equal to the pair representation because the distance from i to j is the same as j to i). then put the individual encoding through a transformer encoder layer, then add another pair encoding to the pair encoding of the individual encoding (created by convolution of the individual representation to create a 7-dimensional representation that is tiled and added so that if this representation at i is $x[i]$, then the pair representation y at i,j is $y[i,j] = x[i] + x[j]$.)

After these layers, the pair encoding is then put through a convolution to convert it from 7D to 1D. This is then put through soft plus, as negative distance doesn't make sense.

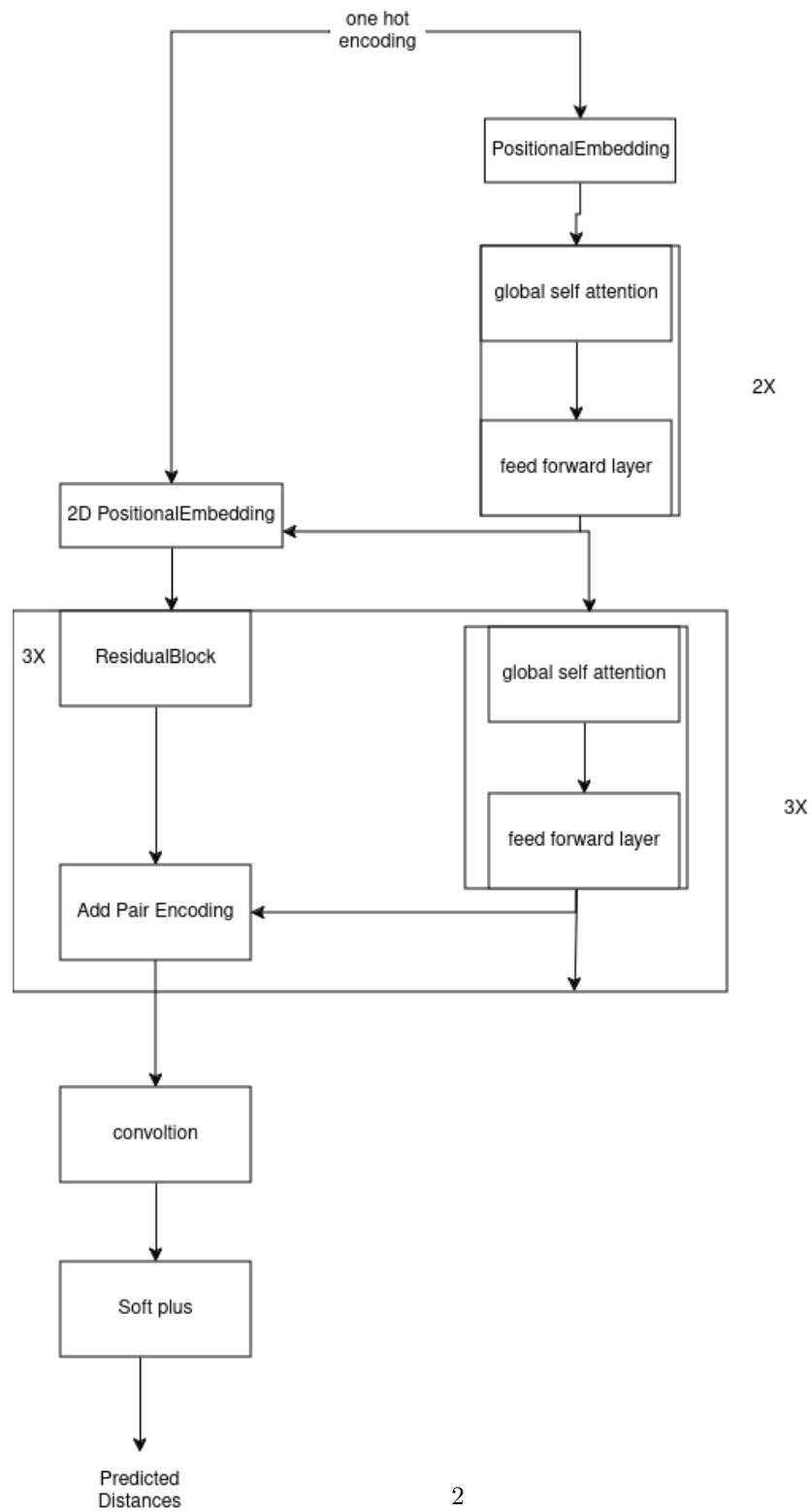


Figure 1: base model data flow diagram

1.2 Change

I added down and up sampling with skips. To do this, I used max down sampling (in a 2 by 2 region to halve the vertical and horizontal height) on the residual blocks (storing the values of pre-down sampling in a stack) until I had done the correct number of down-sampling layers. I then did the rest of the residual blocks normally (and down sampled with max sampling the pair encoding).

I then did the rest of the residual blocks until I only had the number of residual blocks left that was needed to up-sample it. I then did up-sampling sampling with the value before the down sampling added to it, followed by the residual block.

because of the number of residual blocks my network has this means I am testing from 0 to 4 max polling layers.

1.3 Hypothesis

I expected the loss to reduce to with pooling layers, with the best number of pooling layers being the max number of max pooling layers: 4.

I expect to see these results if factors from pairs that are fair away in the pair representation are relevant to the distances between those pairs, and that this information is not lost in the down sampling steps.

that cannot be learnt from and that this information is not lost in the max pool, as well as that even with the max number of pool layers there will still be two residual blocks dealing with all the lower amounts of pools, then the max.

1.4 Method

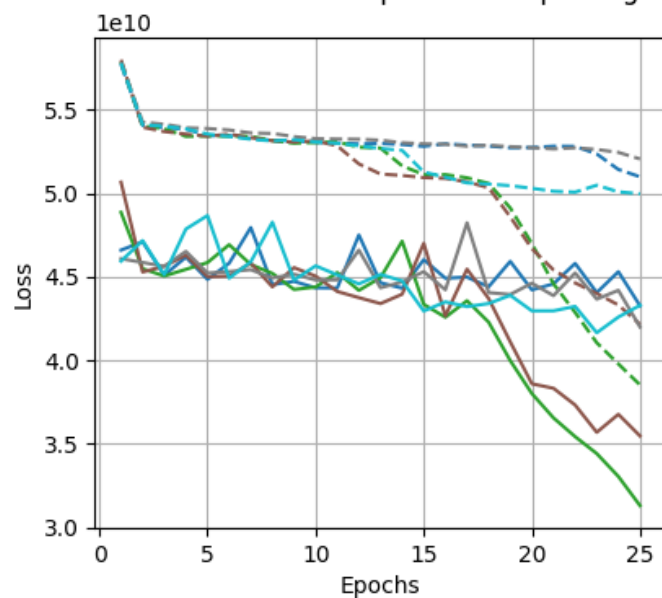
All number of pools were trained for 25 epochs, 5 times on the proved training dataset, while both their total average train performance and their total test performance (tested on the provided test dataset) are recorded for each epoch.

2 Results

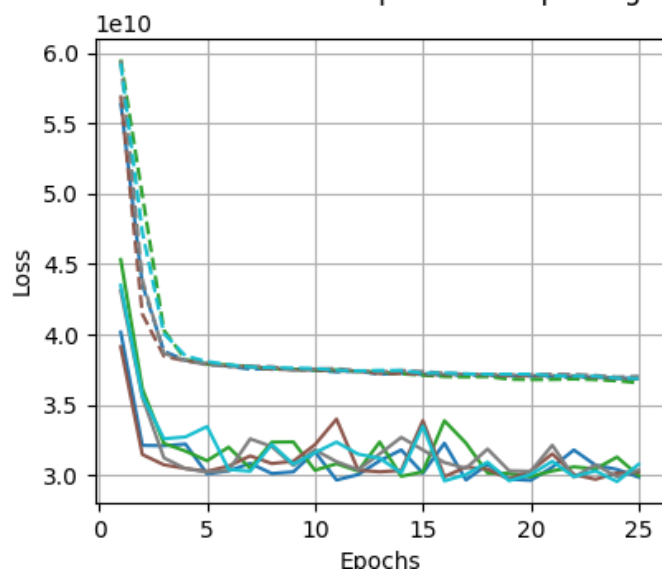
2.1 Graphs of train and test losses

test losses are filled in and train losses are dashed lines.

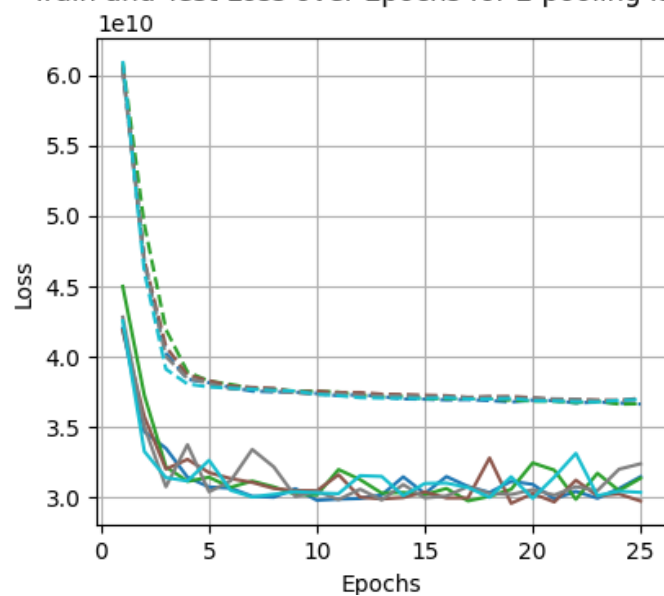
Train and Test Loss over Epochs for 0 pooling layers



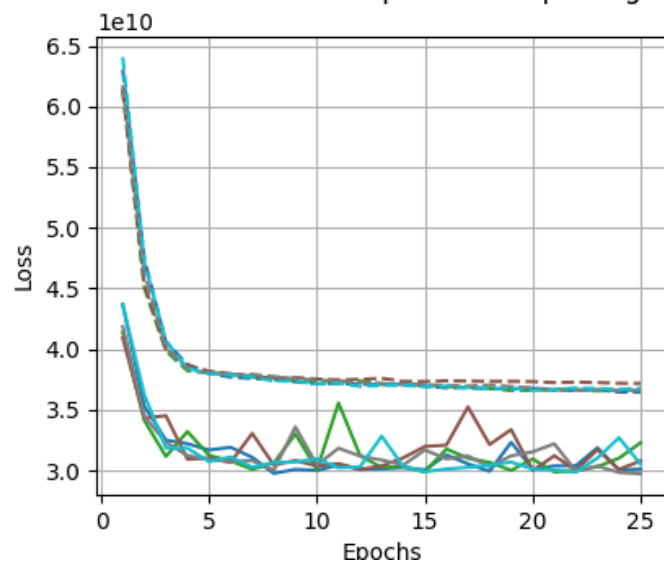
Train and Test Loss over Epochs for 1 pooling layers



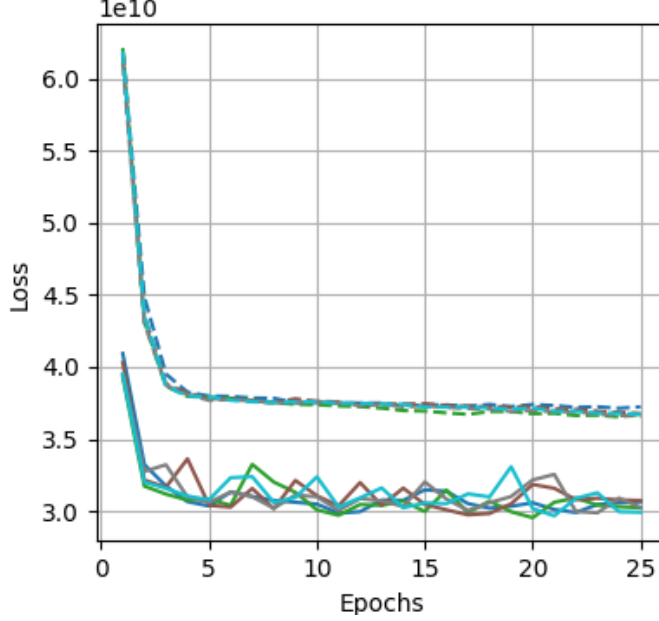
Train and Test Loss over Epochs for 2 pooling layers



Train and Test Loss over Epochs for 3 pooling layers



Train and Test Loss over Epochs for 4 pooling layer



	0	1	2	3	4
0	1.0	1.71e-41	3.4e-40	1.72e-40	8.32e-41
1	1.71e-41	1.0	0.968	0.554	0.567
2	3.4e-40	0.968	1.0	0.543	0.663
3	1.72e-40	0.554	0.543	1.0	0.316
4	8.32e-41	0.567	0.663	0.316	1.0

Table 1: p values for Mann-Whitney u test on verification loss

2.2 Discuss

Adding pool layers seems to reduce the loss significantly, but the number of pool layers added does not seem to affect its effectiveness. This is unexpected. It could be that there is something that is 18 and 30 away in the pair encoding that can be learnt that dramatically reduces loss; it could also be caused by the max pool and up-sampling acting as an extra non-linearity outside of the relu for the pair encoding. It could also be caused by the skip layer, but I use residual adding, so that should not affect the training as they serve the same purpose.

In addition to that, there is weird behaviour in the training for zero pooling layers; it seems that it fell into some local optima, and then randomly some of the training instances start rapidly decreasing both their training and testing loss again well after the initial epochs. This only seems to happen for zero

pooling layers; I don't know what would cause this, so further experiments are needed to find out.

The Mann-Whitney test seems to show that 0 pool layers is definitely different to 1 to 4 poll layers but there is not much certainty in the difference between 1 to 4 compared to 1 to 4 this makes it seem that there are two classes in the of the polling layer further reinforcing the idea that a polling layer existing affects the results but not the number of polling layers.

3 Conclusions

Although, pooling layers make the algorithm more effective the number dose not affect this so, my hypnosis that the max number of pooling layers would be the most effective is not true, I am relatively confident in this statement as I have done a reasonable amount of testing.

3.1 implications

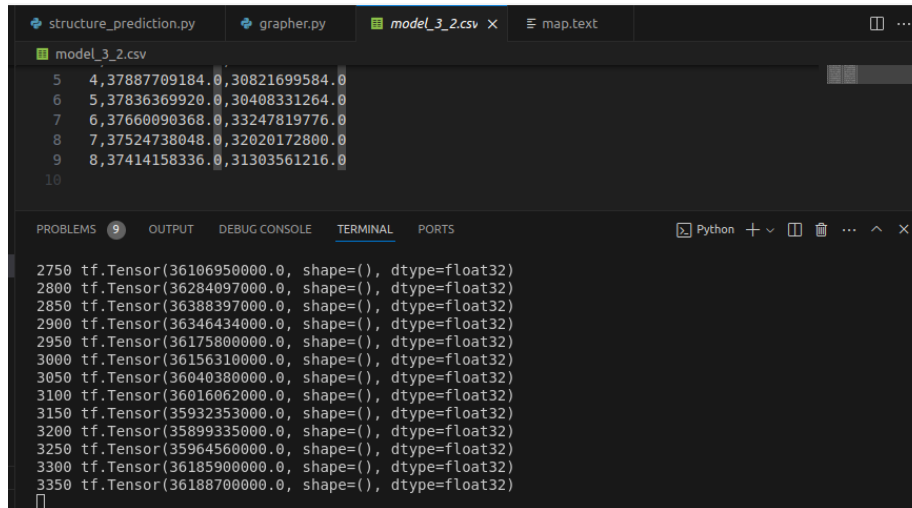
Due to the weird behaviour of my method I am unsure what to take from my results, these results lead me to think that when adding polling layers one should check if pooling layers are achieving there higher success though allowing a larger range of data to be viewed or if they are adding a non-linearity that is affecting the results by allowing the model to learn new functions.

3.2 Future Work

I could go though and test each decision I made in my network such as the pair and individual representation, my use of soft-max or the way I encoded positions and test them. I could also attempt to factor in the evolutionary history of the compound.

I could do more research in to what causes the local optima in 0 pooling layers and allows it to get out of the local optima in some cases, as well as that I could research training the models for longer and see if any of the other models are falling it similar local optima.

4 Screen Shots



The screenshot shows a VS Code editor with two tabs: `model_3_2.csv` and `map.text`. The `model_3_2.csv` tab is active, displaying a CSV file with 5 columns and 10 rows of data. The data is as follows:

	col1	col2	col3	col4
5	4,37887709184.0	0,30821699584.0		
6	5,37836369920.0	0,30408331264.0		
7	6,37660090368.0	0,33247819776.0		
8	7,37524738048.0	0,32020172800.0		
9	8,37414158336.0	0,31303561216.0		
10				

Below the editor, the `TERMINAL` panel is open, showing a list of TensorFlow tensors. The output is as follows:

```
2750 tf.Tensor(36106950000.0, shape=(), dtype=float32)
2800 tf.Tensor(36284097000.0, shape=(), dtype=float32)
2850 tf.Tensor(36388397000.0, shape=(), dtype=float32)
2900 tf.Tensor(36346434000.0, shape=(), dtype=float32)
2950 tf.Tensor(36175800000.0, shape=(), dtype=float32)
3000 tf.Tensor(36156310000.0, shape=(), dtype=float32)
3050 tf.Tensor(36040380000.0, shape=(), dtype=float32)
3100 tf.Tensor(36016062000.0, shape=(), dtype=float32)
3150 tf.Tensor(35932353000.0, shape=(), dtype=float32)
3200 tf.Tensor(35899335000.0, shape=(), dtype=float32)
3250 tf.Tensor(35964560000.0, shape=(), dtype=float32)
3300 tf.Tensor(36185900000.0, shape=(), dtype=float32)
3350 tf.Tensor(36188700000.0, shape=(), dtype=float32)
```