Cosc 440 nki38 96851177

- *EPOCHS: 10 for ALL*
- *BATCH SIZE: 512 for ALL.*
- *This .zip: Report.pdf, Model1.py, Model2.py, results.py which is just MSES over each epoch used for graphing*
- *No .ipynb, as all trained and developed locally.*
- *Also performance seems to vary between runs*

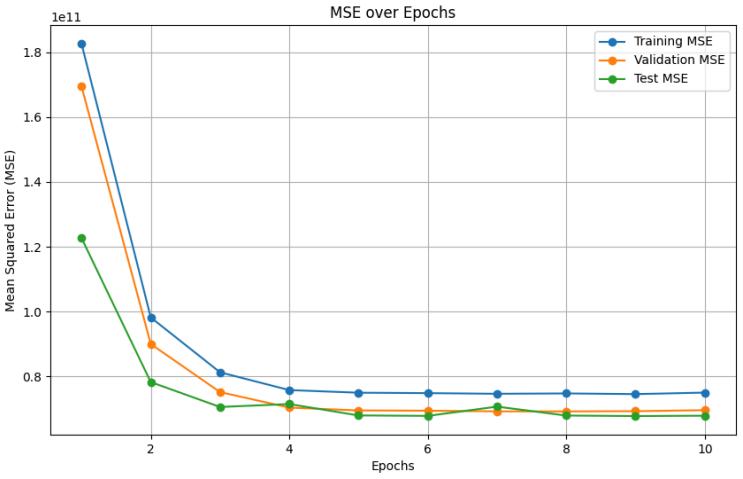
We define a Neural net Model which will predict a 256*256 distance matrix which is the euclidean distance between amino acids in a protein. Our model is trained on a single Nvidia RTX 3080 GPU with ~7gb Memory available, hence reduced batch size and some considerations later.

We consider the evolutionary and primary / primary onehot as our inputs for the model and the tertiary i.e. relative locations) to be the output, which is computed into a distance matrix. We don't perform any masking in the call() method on either model.

Model 1:

1.Architecture

We begin with a **very simple** model for predicting our distance matrix. This is a very simple model; three conv2D layers following concatenated inputs for Evolutionary and Onehot followed by batch normalisation and some reshaping / dense layers to meet our output shape. (full architecture on the final page).



Above; the performance of the initial model over 15 epochs in terms of MSE. We note that the trend *from eyeballing* indicates that more epochs may further reduce MSE.

2. Concerns

Our model is not *particularly* deep; yet it may be liable to some vanishing gradient problem. Our model only captures 'spatial' information in terms of the two inputs; If, say, a relationship exists 'further down' or 'further up' the chain than unseen than it might on the training or even the test data, our model might not produce as useful output. The model should be translationally invariant beyond what conv2d layers can offer. The grid pattern we see could well be a result of overfitting to our training data.

Model 2 / Improvements to model / hypothesis

How can we improve our simple model? Our simple model uses only conv2d (and dense) layers for learning; given the domain for the data, an attention layer is likely a prudent inclusion to our model. Our goal overall is reducing MSE. The following changes (should) reduce MSE on unseen data / the test set here.

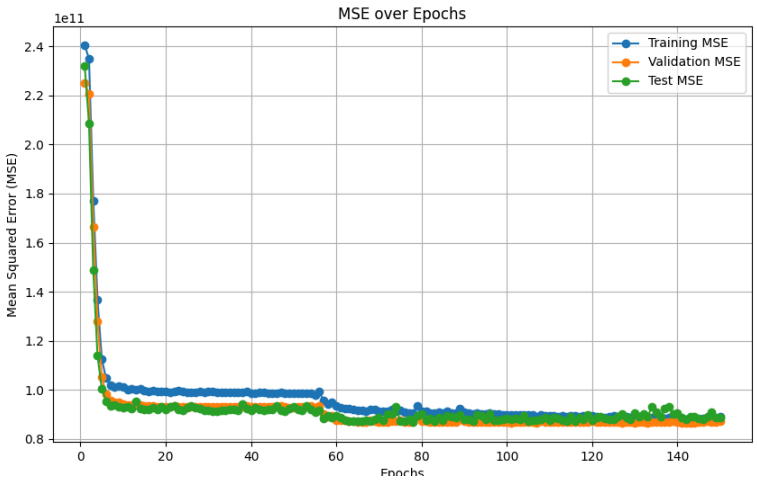
1. Architectural changes

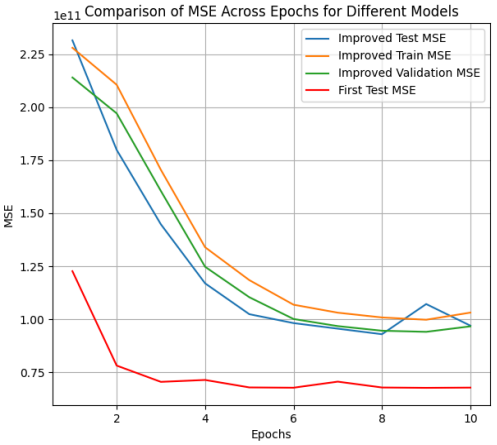
- Introduce Conv2d layers before concatenation with strides to extract local features (2x layers, 32 & 64 filters with 2x2 strides each) from the evo data and one hits respectively. Filters of sizes 32 and 64 in initial layers should capture increasingly complex features from the two ins. (more filters is likely better, but we are memory constrained). (also the introduction of strides should better identify and extract local patterns, which in turn should help with our MSE by better associating features through our back prop)
- Introduce an attention layer after concatenation for determining the most 'important' features in (self attention) the concatenated tensor in terms of itself. This is the most important change we are making.
- Introduce conv layers here which downscale to learn features relative to the attention layer
- (2x layers, 32 & 64 filters with 2x2 strides each)
- Introduce conv layers for upscaling, mostly for fitting the 256*3 final shape, This will also learn features of the data.
- Introduce Dropout Layers, to hopefully reduce overall MSE on test and val.
    - We do these before the final dense layer and after the attention layer. Ideally shrinking reliance on attention derived features and also network-as-a-whole features; assisting to better generalise.
- Introduce layers which mitigate potential vanishing gradient (leaky relu), to hopefully reduce overall MSE on test and val.
- Learning rate remains unchanged. Batch Size remains unchanged to fit in memory (although given the domain, larger batch sizes may help the learning process).

Our overarching expectation is that these architectural changes, primarily the introduction of further convolutional layers with strides, filters and the attention layer will substantially reduce MSE on new data. LeakyReLU and dropout layers will also achieve this, being overfitting reduction layers which also contribute to better stabilising the trianing process. (ideally mitigating the grid effect we see in simple model).

Attention, we'd best expect to see effects on longer sequences, but should still see improvement on 256-limited.

2. Results of Changes





On the left we see the result of running the improved model over 150 epochs. We capture the change in test and validation error over that time. We show in the graph on the right, the comparison between all sets of the improved model (test, train, val) against the test set in the base model. We see that the simple model 'settles' quicker (per table below) to it's ~final mse for test. With time the improved model does achieve improved MSE.

The full break down is below:

Notably; we see lower overall MSE on the improved model; after only **crossing point**.

The following table shows some comparison on *25* epochs of each part of our above ideas; including models with no dropout, the base model and a model using relu and not leaky relu.

	Simple	**Improved**	**Improved no Dropout**	**Improved no L.Relu**
Test MSE (10 epochs)	11591602995 2.0	82842796032.0	84104232960.0	89558556672.0
Test MSE (25 epochs)	11586219212 8.0	81730543616.0	82805809152.0	87410065408.0
Validation MSE (10 epochs)	11621634867 2.0	84886630400.0	85756379136.0	88886930195.6923
Validation MSE (25 epochs)	11636489420 8.0	83543984915.69 23	84978775591.3846 1	86818993388.3077
Improvement over simple.	0%	28.19%	26.98%	25.4%
Test MSE (final)	11586219000 0	81730540000	82805810000	87410065000
Epoch at Min Validation MSE	21	25	24	24
Avg. Val MSE (last 5 epochs)	11625218728 5.66154	83953332980.18 462	85094417455.2615 4	87173531742.52309
Improvement Over Simple (lowered MSE)	0%	26.8%	25%	27.76%

We also note that for all MSE's for test, the improved set is significantly lower; per the graph above it is clear that at no point in (25 epochs) the MSE for simple is lower than our improved model; thusly our results validate our hypothesis (or, rather the amalgamation of hypotheses we made).

Conclusion
Our major changes to the model have positively impacted MSE on the test set positively (note that we're using the same mix of test / train / val, and that performance on one combination of these isn't representative of any other given performance.

Further work:
A self attention model on the one hots does still make sense given the domain (i.e. the sequence of the amino acids informs the overall att. model) but it would likely improve performance to, instead, implement some embedding. (for ease of processing / tensor fitting, we didn't). Were memory constraints less *imposing*, training deeper model with multihead attention layers, independently training parts of the model and joini

A flaw with this model is the *complexity* and the limited hardware; Given the domain training on larger batches would also be beneficial. Adding multihead attention between the separate input tensors would likely be hugely beneficial in identifying relationships based on the protein sequence itself onto the evolutionary data (or vica versa) in determining the most appropriate features, but with memory blowout, this was not achieved.

We also did not explore optimizers, algorithmic architectural design, hyperparameter tuning etc etc. Many of our decisions (i.e. # of filters) were predicated on the memory footprint again.

We should also explore embedding layers for further feature extraction enhancements / improvements.

Something we noted *after* filling in the results table, was that each time the NN is trained, the MSE is different. We neglected to set seeds for adam or batch ‘shuffling’. Doing so would address issues in our results being inconsistent. The initial vector of weights is also fully random.

Implications.
We showed that adding attention (and conv layers) significantly helps in highlighting important features through the performance of our model. We can point to any part of the model as a point of reference for implications. Researchers should allocate funds to multi GPU’s with *easy* memory scaling.

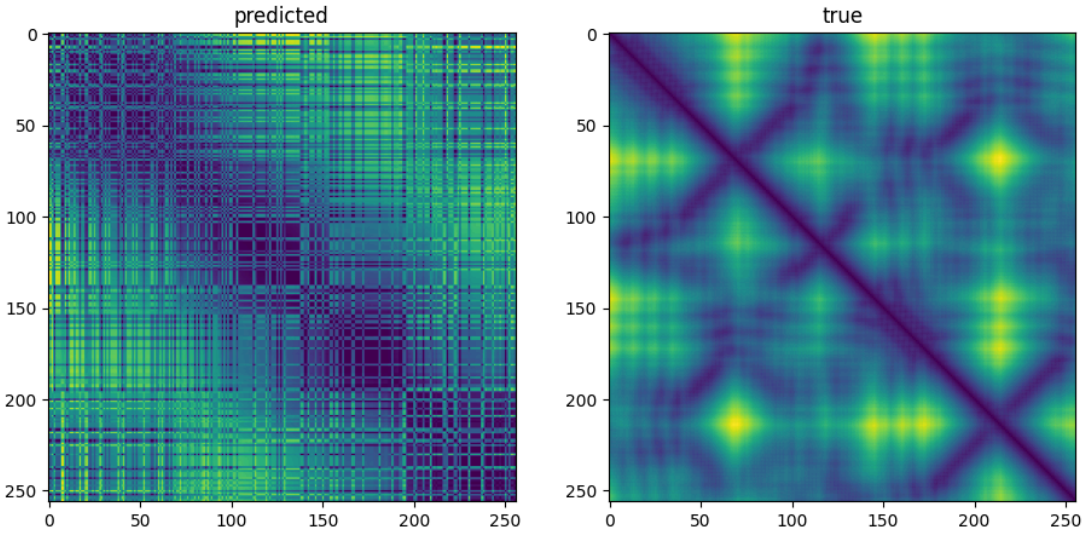
—

:Architectures:

SImple Model:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(512, 256, 21, 16)	304
conv2d_1 (Conv2D)	(512, 256, 21, 32)	4,640
conv2d_2 (Conv2D)	(512, 256, 21, 16)	4,624
batch_normalization (BatchNormalization)	(512, 256, 21, 16)	64
flatten (Flatten)	(512, 86016)	0
dense (Dense)	(512, 768)	66,061,056

Total params: 66,070,688 (252.04 MB)
Trainable params: 66,070,656 (252.04 MB)
Non-trainable params: 32 (128.00 B)



Improved Model:

Layer (type)	Output Shape	Param #
attention (Attention)	(512, 256, 159)	0
conv2d (Conv2D)	(512, 128, 11, 32)	320
conv2d_1 (Conv2D)	(512, 64, 6, 64)	18,496
conv2d_2 (Conv2D)	(512, 127, 10, 32)	320
conv2d_3 (Conv2D)	(512, 63, 4, 64)	18,496
max_pooling2d (MaxPooling2D)	?	0
conv2d_4 (Conv2D)	(512, 256, 53, 21)	588
conv2d_5 (Conv2D)	(512, 256, 53, 10)	1,900
conv2d_6 (Conv2D)	(512, 256, 53, 3)	273
conv2d_7 (Conv2D)	(512, 256, 53, 3)	12
dense (Dense)	(512, 256, 53, 3)	12
dense_1 (Dense)	(512, 256, 3)	480
dropout (Dropout)	?	0
dropout_1 (Dropout)	?	0
leaky_re_lu (LeakyReLU)	?	0
leaky_re_lu_1 (LeakyReLU)	?	0
leaky_re_lu_2 (LeakyReLU)	?	0
leaky_re_lu_3 (LeakyReLU)	?	0

Total params: 40,897 (159.75 KB)
Trainable params: 40,897 (159.75 KB)
Non-trainable params: 0 (0.00 B)