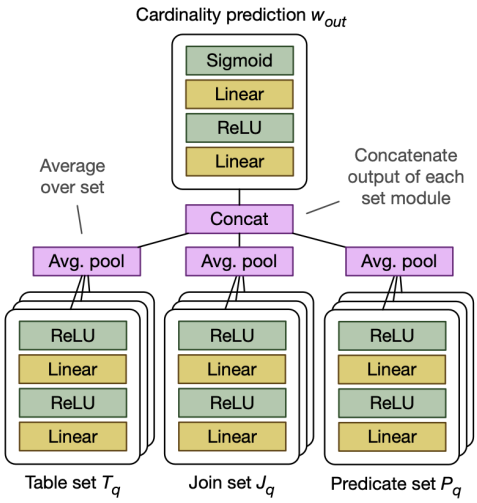
Optimizing the ANN Architecture for Learned Cardinalities

Raksha Ramesh

This project is derived from the paper in the readings titled: [Learned Cardinalities: Estimating Correlated Joins with Deep Learning.]
In this paper, the authors propose a deep learning solution to estimating the cardinalities of the joins before feeding it into the query optimiser. The model they employ is an MSCN (multi-set convolutional network), a neural network module for operating on sets.
In this project, I experiment with different hyperparameters and find the following configuration to give the best performance:

1. **Architecture - Encoder-decoder for the 3 MLPs - table, predicates and joins.**
2. **Batch Size - 128 Samples**
3. Hidden Units - varies per layer - but starts with 256 and builds in powers of 2.
4. Learning Rate - 0.001
5. **Loss Function - Mean Logarithmic Squared Error (MLSE)**
6. Optimiser - Adam

Author's Architecture:



The input to the MSCN is a representation of a query - denoted in terms of multiple sets(specifically 3 - one for the tables, one for the joins and one for the predicates). A simple 2 layer MLP is learnt per set in the input. The output of each set is the average over the individual transformed representations of its elements. Then the individual representations of each of the sets are concatenated and passed through another output MLP.

Experimental Setup:

The authors train and test their model on a NVIDIA GeForce GTX 1050 Ti (4 GB GDDR5) GPU using the PyTorch framework. They use 100,000 random queries with a 90-10 train-test split. The Hyperparameters the authors have set:

1. Batch Size - 64 Samples
2. Hidden Units - 256
3. Learning Rate - 0.001
4. Loss Function - Q-Error
5. Optimiser - Adam

I used this configuration and ran a baseline test on the synthetic workload as my system configuration varies with that of the authors, and it would be better to compare the results of my testing with my own run of the baseline rather than compare with the numbers quoted by the paper.

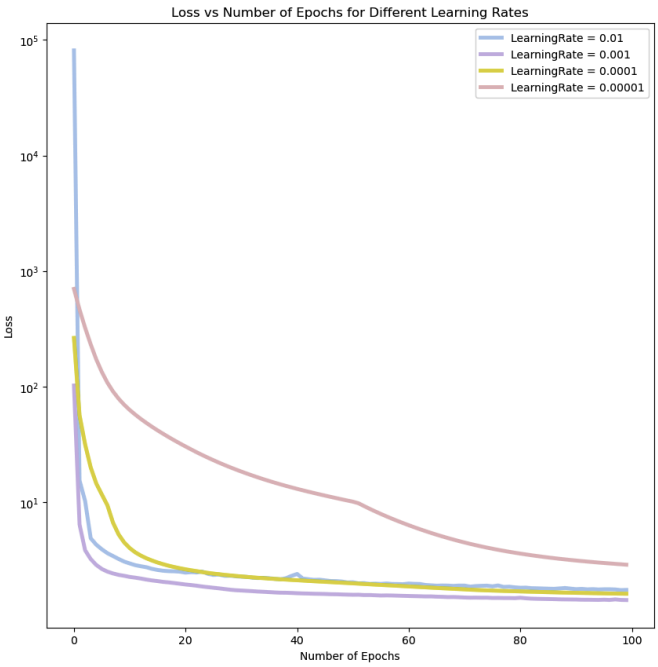
I obtained the following results (Q-Error) for the baseline while running it on the synthetic workload:

Median	90th	95th	99th	Max	Mean	Training time per epoch
1.22	3.85	7.7	31.0	1809	3.12	13.1

Experiments Run:

Experimental setup:
Training and testing was done on Google Colab, using the T4 (Tesla T4) GPU using the PyTorch framework.

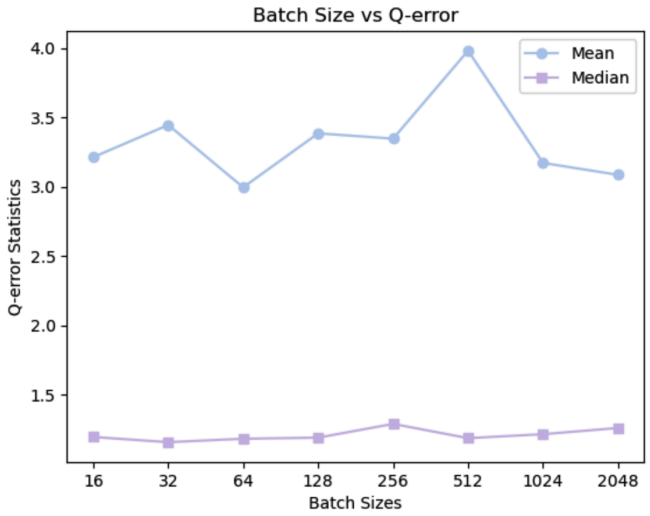
1. Learning Rates

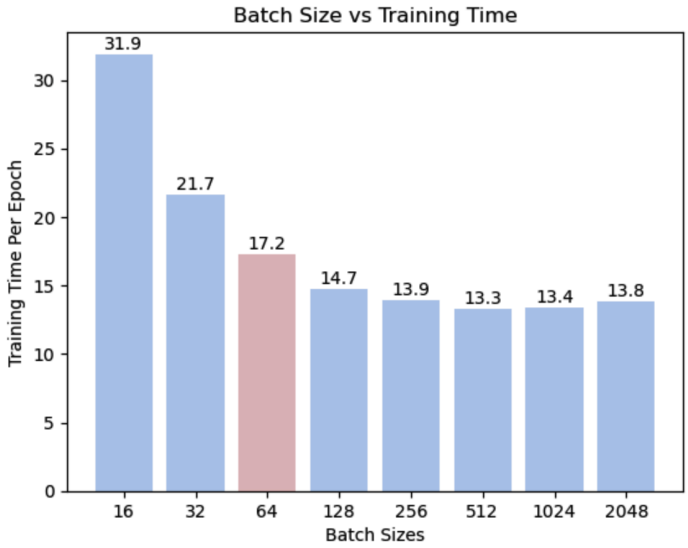


I first began the testing with different learning rates. I tried rates from 10^{-1} to 10^{-5} in negative powers of 10 [1].
In the plot, I have used a symmetrical logarithm scale for the vertical axis, with the linear range set to 10. So, values from 0 to 10 are on a linear scale and all values above 10 are on a logarithmic scale (The same scale has been used for all Loss vs Epochs graphs henceforth).

It can be seen that the purple graph, corresponding to the learning rate of **0.001**, achieves a faster convergence and converges to a lower overall loss than the other learning rates.

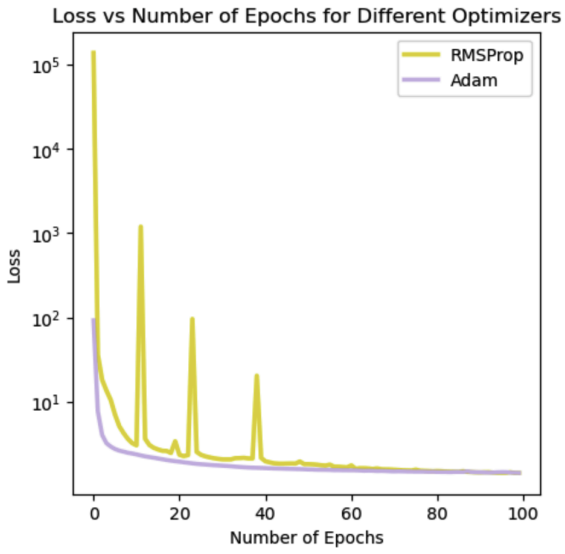
2. Batch Size





I tested batch sizes in powers of 2 starting from 2^4 up till 2^{11}. I have plotted the q-error for the different batch sizes as well as their corresponding training time per epoch. The use of small batch sizes has been shown to improve generalization performance and optimization convergence, whereas large batch sizes improve the parallelism and hence reduce the time taken per epoch [2]. This can be seen in the graphs above.
While a batch size of 64 achieves the best performance with respect to the mean, comparing both mean and median performance along with the corresponding training time per epoch, I believe a batch size of **128** would be a better choice.

3. Optimizers

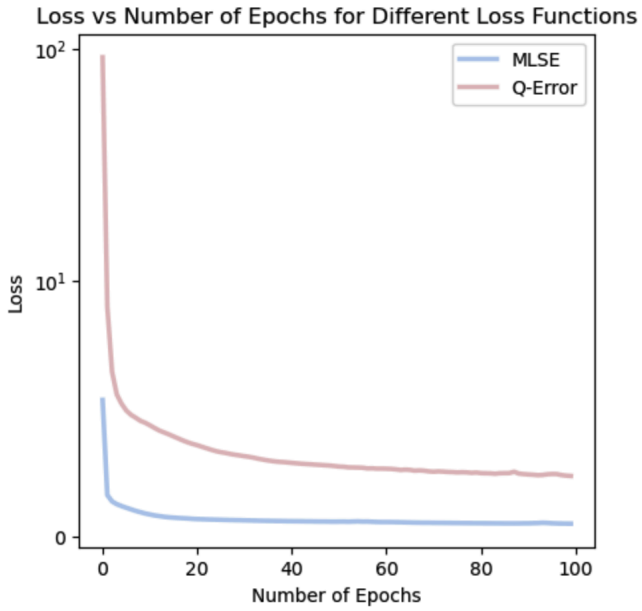


I compared the following optimizers: Adam, Stochastic Gradient Descent (SGD), SGD with momentum (different values), SGD with Nesterov acceleration, and RMSProp. Out of these five, only RMSProp and Adam converged to an acceptable loss.

As it is visible in the graph, despite RMSProp and Adam converging to similar losses, **Adam** does so very smoothly and quickly.
In the first 5 epochs, Adam noticed a significant drop over 90% in the loss, whereas a lot of fluctuation was observed in RMSProp, delaying the convergence.

4. Loss Functions

The authors of the paper mention that they tested Mean Squared Error(MSE) in addition to the Q-Error during their training process. They however pick the Q-Error as their loss function as it directly corresponds to the objective they are trying to optimize.



Testing this out on my own, using MSE as the loss function had losses and performance that was significantly worse (the mean of the predictions on the synthetic workload was more than 100x worse) than while using Q-Error.
I noticed that the main cause for this discrepancy was the fact that the cardinality estimates are generally wrong by *magnitudes* when compared to the actual values of the cardinalities, hence, choosing a loss function that works on this difference becomes imperative.
I applied the MSE loss function over the predicted and true values of the cardinalities *after taking their log* **(MLSE)**.

Loss	Median	90th	95th	99th	Max	Mean
MLSE	1.21	3.55	7.33	35.61	1170	2.92
QErorr	1.22	3.85	7.7	31.0	1809	3.12

When testing on the synthetic workload (still measuring the Q-Error, despite having a different loss function), the **model trained using MLSE performs better than the model trained using the Q-Error loss function** on most metrics.

5. Activation Function

Since ReLU offers better and quicker convergence than activation functions like sigmoid and tanh, I only tested out with leaky-ReLU, which didn't offer any performance gain. This was the performance (Q-Error) obtained on the synthetic workload:

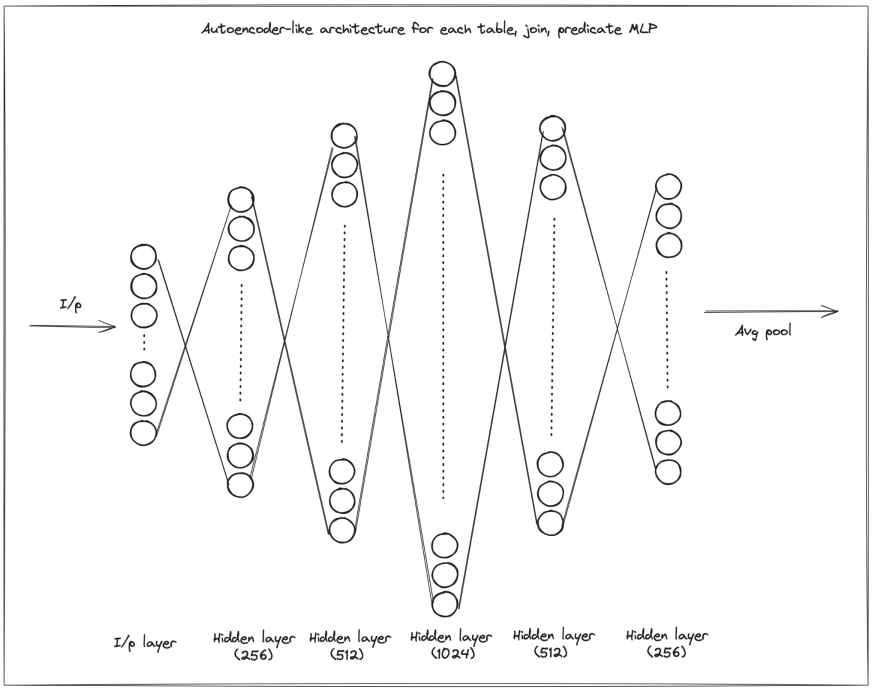
Activation Function	Median	90th	95th	99th	Max	Mean
Leaky-ReLU	1.25	3.72	7.48	33.29	1327	3.06
ReLU	1.22	3.85	7.7	31.0	1809	3.12

6. Architectural Changes

The first change I made was to test the performance of the model with **another hidden layer** in the MLPs corresponding to the tables, joins and predicates. I reduced the number of hidden units by half, and created 2 layers. I then also tested with **3 full layers.** There was **no performance gain** in any of the metrics for any of the different architectures.

Since the inputs (tables, joins, predicates) are representations of features, I reasoned that having an **autoencoder-like structure for each of the MLPs** would be worth considering.
I also considered a version where I added the **encoder-decoder to the output MLP**, but that offered **very little gain in performance.**

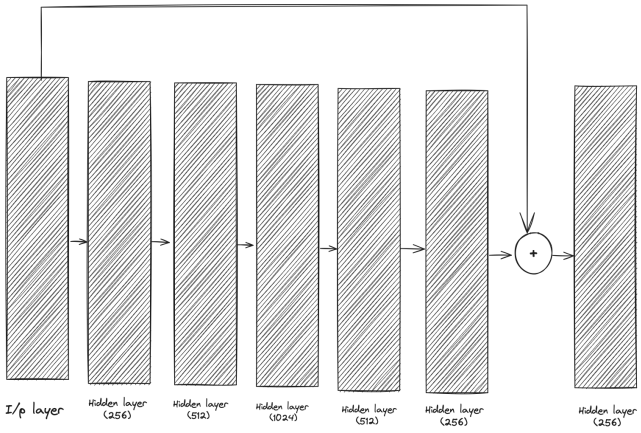
The architecture for the predicate, join and table MLPs I tested was:



There was a **big drop in the max Q-error** of the new architecture as compared to the baseline. Which lends itself into the **reduction of the mean as well.** This hints that the number of outlier predictions (really bad estimations) have reduced in the new architecture.

Architecture	Median	90th	95th	99th	Max	Mean
Autoencoder-like	1.21	3.68	7.78	34.81	549	2.81
Baseline	1.22	3.85	7.7	31.0	1809	3.12

I also tested out adding a **residual connection** to increase the power of the network as it had potential for a better performance and wasn't currently overfitting too much (mean Q-Error for training was 2.43 and for the validation set it was 2.81). I experimented both **with and without dropout layers** reasoning that due to the added complexity, the model might start to overfit**.**



There was however **no significant gain** in performance in this case either.

Conclusion

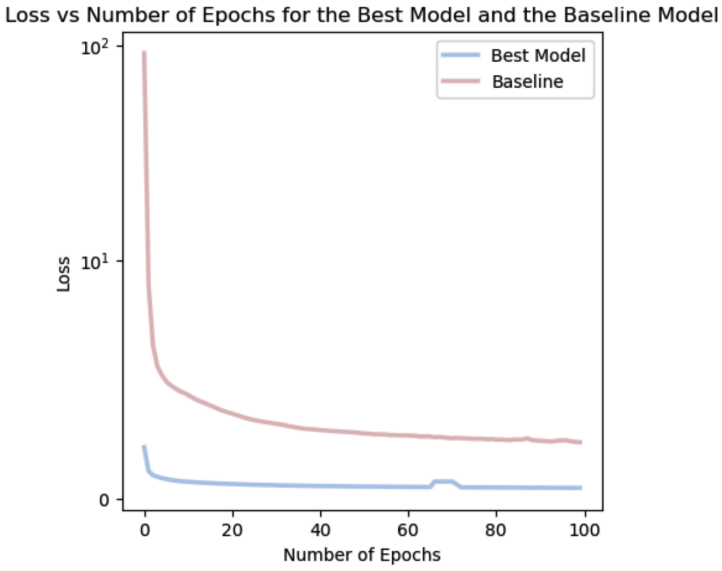
After all the experiments, the following model should be the most optimal model for the use case of cardinality estimation.

7. **Architecture - Encoder-decoder for the 3 MLPs - table, predicates and joins.**
8. **Batch Size - 128 Samples**
9. Hidden Units - varies per layer - but starts with 256 and builds in powers of 2.
10. Learning Rate - 0.001
11. **Loss Function - Mean Logarithmic Squared Error (MLSE)**
12. Optimiser - Adam

Testing the predictions of this complete model on the synthetic workload and comparing it with the base model provided the following results for the Q-Error:

Model-Config	Median	90th	95th	99th	Max	Mean
Best-Model	1.16	3.41	7.06	34.5	1496	2.98
Baseline	1.22	3.85	7.7	31.0	1809	3.12

It is seen that the model beats the baseline on all fronts except for the 99th percentile metric, considering that the Max metric is lesser than the baseline, this can be suggestive of a more even spread and reduction in outliers.



Seen from the loss vs number of epochs graph, owing to the MLSE loss function, the best model achieves a drop faster and converges quicker.

In conclusion, the new model with the selected hyperparameters beats the model suggested in the paper by a very small margin. Even then, the mean predictions are still wrong by the ratio of almost 3. The ideal q-error should be close to one, so that the difference between the predicted and actual cardinalities can be measured in a linear scale.

This is a testament to how difficult the task cardinality estimation truly is and solving this problem may require a more complex neural network (maybe transformers instead of bitmaps?) or an entirely new approach to the problem as seen in [3] and [4] with tree ensemble models.

References

1. Page 434, Deep Learning, 2016.
2. Revisiting Small Batch Training for Deep Neural Networks
3. Cardinality estimation with a machine learning approach
4. An Empirical Analysis of Deep Learning for Cardinality Estimation