COL775
Assignmnet 1

Sanket Gandhi

March, 2023

1 Model Links

The links are as follows

- Part 1.1 link
- Part 1.2 link

2 Resnet

Part 1.1.1

Generic model which take n and r as input is in the

```
src/model.py
```

The model can be trained by running the command

```
python main.py --task train --args arg0 val0 arg1 val1 ..
```

After $--args$ the script expects *argumnet name* followed by its *value*. For example

```
python main.py --task train --args n 2 r 10 experiment_name demo
```

The possible *args* are in *config.py*. The *train.data_path* is the list so it can only be changed from *config.py*.

Part 1.1.2

A lot of different variations of optimizers and learning rate decay were tried. Below are the results and findings. For all the experiments $n = 2$ and $r = 10$ were used and the batch size was 128.

Optimizers

The experiments were done with SGD, SGD with momentum, RMSprop, Adam, Adagrad. Fig 1 shows training loss for the model when trained with different optimizers with $lr = 0.1$ and $lr = 0.01$. From the plots, it is clear that for higher learning rates SGD, SGD with momentum, and Adagrad perform significantly better than Adam, RMSprop. For $lr = 0.01$ the trend is almost the same but Adam and RMSprop shows some improvement. The learning rate was constant for the whole training. This shows that even Adam, RMSprop uses heavy inductive biases they are not always better than simple optimizers like SGD. One of the reasons for such behavior might be the high bias for the initial values of the gradient by RMSprop and Adam which is just zero. As Adagrad does not have a such high bias to the initial gradient it does well.

Learning Rate Decay

The experiments were done with exponential, step, linear, and constant learning rates. Fig 2 shows training loss for the model when trained with SGD and different decay rates. The same trend is observed with other optimizers. It is evident from the plots that the exponential decay rate performs better than other lr decay schemes.

Intial learning rate

Initial learning rates have been observed to affect the accumulative base optimizers Adam, RMSprop, and Adagrad very significantly than simple optimizers like SGD, SGD with momentum. The reason is same initial strong priors about gradient are penalized highly with a high learning rate. More plots are in the appendix. The accuracy is also provided in the appendix.

Part 1.1.3

For early stopping, I implemented early stopping with a patience algorithm $p = \{1, 2, 3, 4, 5\}$. All the optimizers with possible learning rate decay were experimented with. Fig 3 shows the full error curves for the validation and train loss for SGD with momentum and SGD with exponential weight decay and starting $lr = 0.1$.
Now let's see the validation curves for different p. Fig 4, Fig 5, Fig 6, Fig 7, 8 show the validation and train loss for different patience. Also, Table 1 summarizes the results. Now I only got good results for Adagrad, SGD and SGD with momentum with exponential lr decay or step lr decay and starting $lr = 0.1$. Ideally, the test accuracy should increase as the patience increase but as the initially, parameters of the model are sensitive to initialization for low values of patience we get inconsistent results. As we are only allowed to take patience one I will be submitting the Adagrad with constant lr and 1 patience as the final model.

Model	Train Data			Val Data			Test Data		
	F1 micro	F1 macro	Acc	F1 micro	F1 macro	Acc	F1 micro	F1 macro	Acc
adag-con-P1	0.801	0.802	0.801	0.801	0.802	0.801	0.793	0.795	**0.793**
sgdm-con-P1	0.775	0.773	0.775	0.775	0.773	0.775	0.77	0.768	**0.77**
adag-exp-P2	0.913	0.913	0.913	0.913	0.913	0.913	0.898	0.898	**0.898**
sgdm-exp-P2	0.887	0.888	0.887	0.887	0.888	0.887	0.876	0.877	**0.876**
adag-exp-P3	0.855	0.856	0.855	0.855	0.856	0.855	0.846	0.846	**0.846**
sgdm-exp-P3	0.875	0.875	0.875	0.875	0.875	0.875	0.864	0.864	**0.864**
adag-exp-P4	0.899	0.898	0.899	0.899	0.898	0.899	0.886	0.885	**0.886**
sgdm-ste-P4	0.879	0.879	0.879	0.879	0.879	0.879	0.868	0.868	**0.868**
adag-exp-P5	0.881	0.88	0.881	0.881	0.88	0.881	0.866	0.865	**0.866**
sgdm-exp-P5	0.907	0.907	0.907	0.907	0.907	0.907	0.894	0.894	**0.894**

Table 1: Train, val, and test performances of models for different patience values

Part 1.2

All the implementations of normalization can be found in the

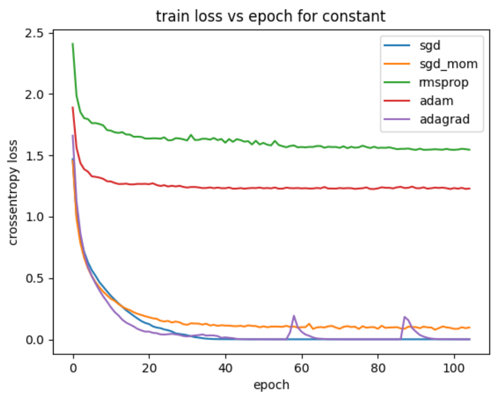
`model.py`

file.

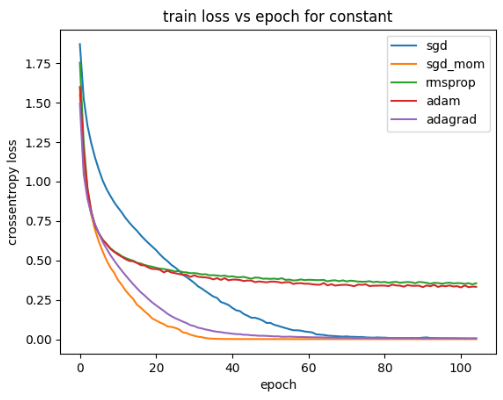
Part 1.2.1

The normalization can be specified as the argument to model *init* function with

`norm = [inbuilt, bn, gn, ln, bin, in, none]`

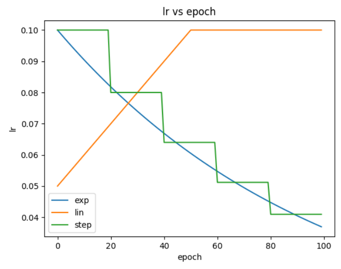


(a) $lr = 0.1$

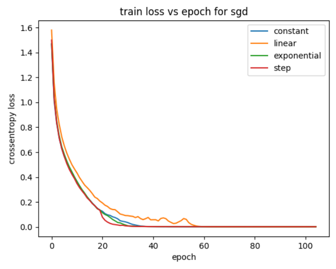


(b) $lr = 0.01$

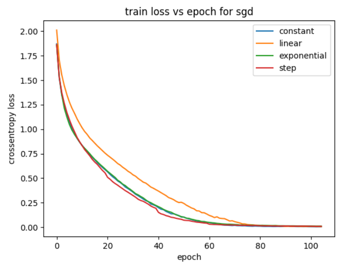
Figure 1: Training loss for different optimizer



(a) lr decay



(b) $lr = 0.1$



(c) $lr = 0.01$

Figure 2: Training loss for SGD with different decay rate

file. I followed the implementation from respective papers and used the default parameters as seen in their official implementation or PyTorch implementation.

Part 1.2.2

I trained all 6 models for a total 100 epochs with *Adagrad* optimizer and $lr = 0.1$ with exponential weight decay. The models can be trained as

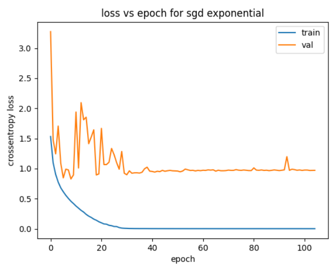
```
    python main.py --task train --args normalization ln optimizer Adagrad lr_scheduler.name
exponential root <experiment_root_folder> experiment_name <experiment_folder_name>
```

Part 1.2.3

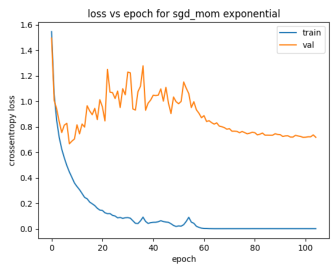
The loss curve for all 6 models can be seen in Figure 9. As mentioned in the paper layer normalization does not perform well on CNN and is evident from loss curves.

Part 1.2.4

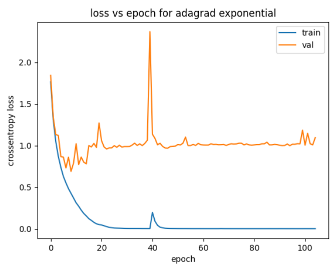
Figure 10 shows the implemented batch normalization and the inbuilt one. It is evident from the loss curve that both perform almost identically.



(a) SGD

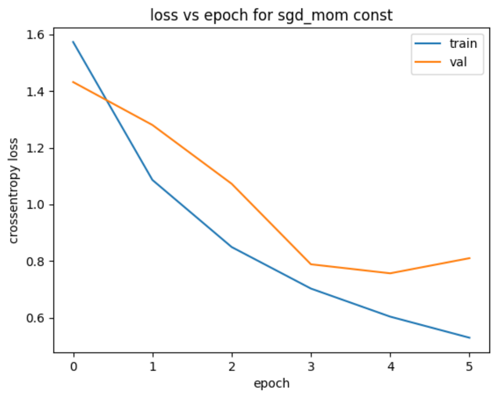


(b) SGD with momentum

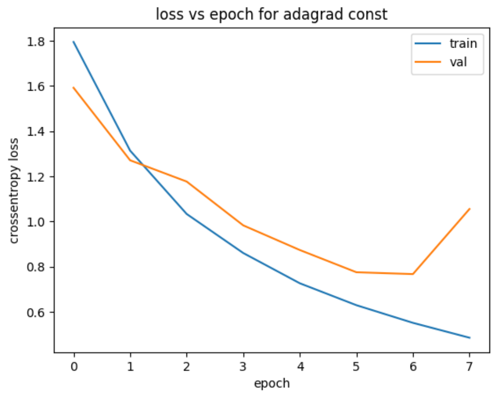


(c) Adagrad

Figure 3: Train and val loss for with exponential weight decay and $lr = 0.1$

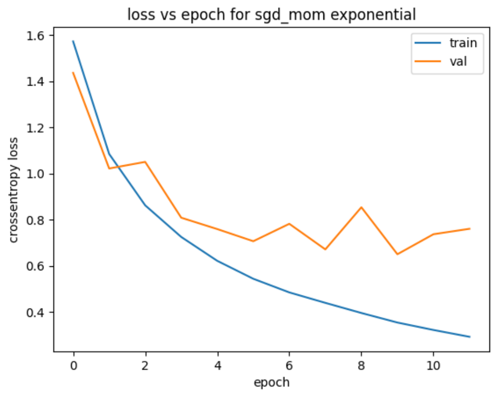


(a) SGD with momentum constant lr

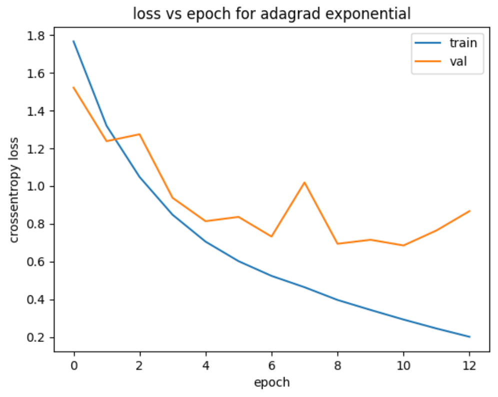


(b) Adagrad constant lr

Figure 4: Train and val loss for $patience = 1$

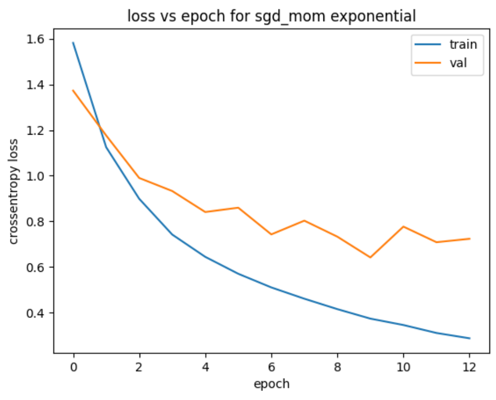


(a) SGD with momentum exponential lr decay

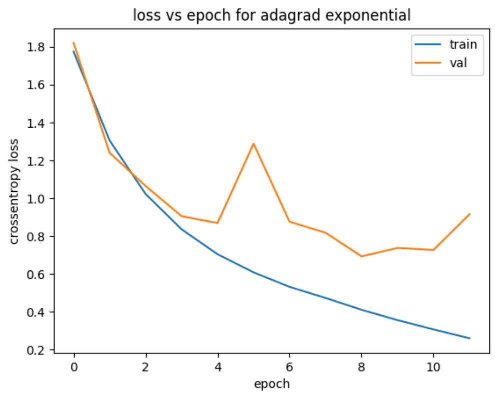


(b) Adagrad exponential lr decay

Figure 5: Train and val loss for $patience = 2$

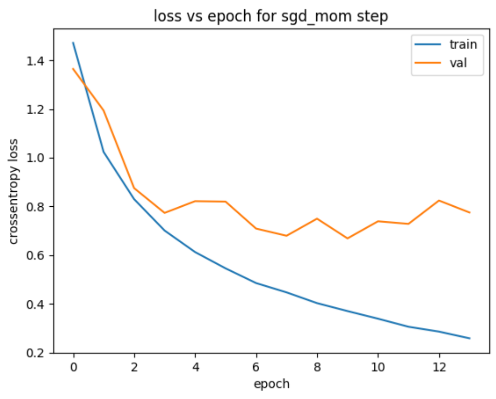


(a) SGD with momentum exponential lr decay

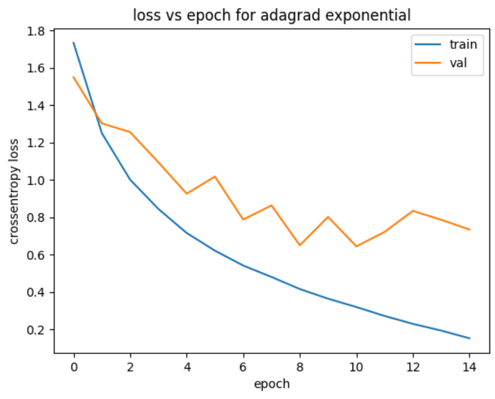


(b) Adagrad exponential lr decay

Figure 6: Train and val loss for $patience = 3$

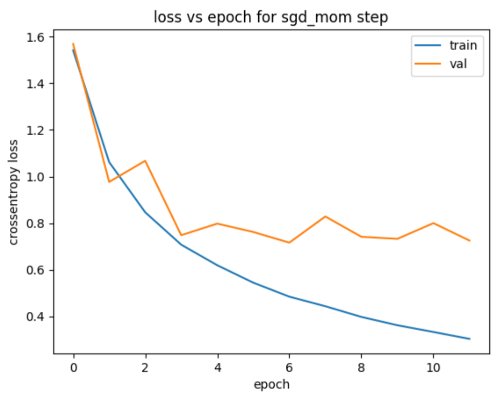


(a) SGD with momentum step lr decay

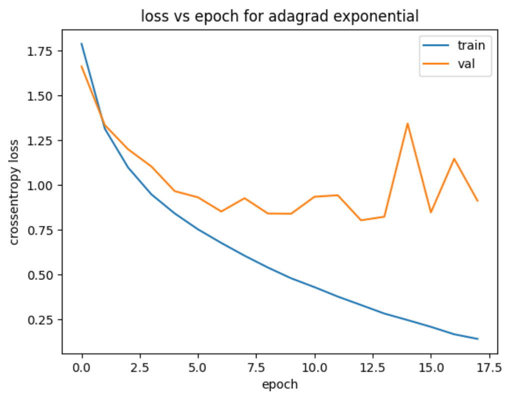


(b) Adagrad exponential lr decay

Figure 7: Train and val loss for $patience = 4$



(a) SGD with momentum step lr decay



(b) Adagrad exponential lr decay

Figure 8: Train and val loss for $patience = 5$

Part 1.2.5

Table 2 shows the detailed results of different normalizations. The optimizer is Adagrad and exponential weight decay was used. The batch size was 128 and the initial lr was 0.1. Three different patience also tried $\{1, 2, 3\}$. In all the patience value batch instance normalization performs well. The model with no normalization performs worse in all scenarios. Group normalization also performs badly in almost all patience values. Figure 11, 12 and 15

Part 1.2.6

Figure 14 shows the detailed results of different batch sizes for group normalization and batch normalization. From curve, it is somehow evident that group normalization performs better on lower batch sizes than the batch norm. From the quantitative perspective at batch size 8 group norm got 4 percent less accurate than it's 128 batch variant where as batch normalization got 8.3 percent lower accuracy than its 128 batch variant after 25 epochs.

Part 1.2.7

Figure **??** shows the quantile features of the various normalizations. A model with no normalization is having no feature detection. But instance normalization shows high feature detection values of all other normalization. The poor performance of layer norm is also supported by its poor feature detection. Inbuilt and Batch norm shows a similar trend as expected.

Model	Train Data			Val Data			Test Data		
	F1 micro	F1 macro	Acc	F1 micro	F1 macro	Acc	F1 micro	F1 macro	Acc
none-P1	0.1	0.018	0.1	0.1	0.018	0.1	0.1	0.018	0.1
inbuilt-P1	0.723	0.72	0.723	0.723	0.72	0.723	0.719	0.715	0.719
ln-P1	0.4	0.383	0.4	0.4	0.383	0.4	0.4	0.383	0.4
bn-P1	0.77	0.869	0.77	0.869	0.768	0.869	0.758	0.858	0.758
gn-P1	0.598	0.596	0.598	0.598	0.596	0.598	0.595	0.593	0.595
in-P1	0.8	0.798	0.8	0.8	0.798	0.8	0.793	0.791	0.793
bin-P1	0.82	0.819	0.82	0.92	0.819	0.82	0.804	0.803	**0.804**
none-P2	0.101	0.018	0.101	0.101	0.018	0.101	0.1	0.018	0.1
inbuilt-P2	0.839	0.837	0.839	0.839	0.837	0.839	0.831	0.828	0.831
ln-P2	0.505	0.493	0.505	0.505	0.493	0.505	0.502	0.489	0.502
bn-P2	0.874	0.873	0.874	0.873	0.873	0.873	0.862	0.861	0.862
gn-P2	0.703	0.698	0.703	0.703	0.698	0.703	0.698	0.693	0.698
in-P2	0.878	0.878	0.878	0.878	0.878	0.878	0.865	0.865	0.865
bin-P2	0.886	0.885	0.886	0.886	0.886	0.886	0.874	0.873	**0.874**
none-P3	0.099	0.018	0.099	0.099	0.018	0.099	0.099	0.018	0.099
inbuilt-P3	0.884	0.886	0.884	0.884	0.886	0.884	0.871	0.873	0.871
ln-P3	0.508	0.494	0.508	0.508	0.494	0.508	0.506	0.492	0.506
bn-P3	0.872	0.871	0.872	0.871	0.871	0.871	0.861	0.86	0.861
gn-P3	0.811	0.808	0.811	0.811	0.808	0.811	0.801	0.798	0.801
in-P3	0.891	0.891	0.891	0.891	0.891	0.891	0.875	0.876	0.875
bin-P3	0.918	0.918	0.918	0.918	0.918	0.918	0.904	0.904	**0.904**

Table 2: Train, val, and test performances of models with different normalizations for different patience values

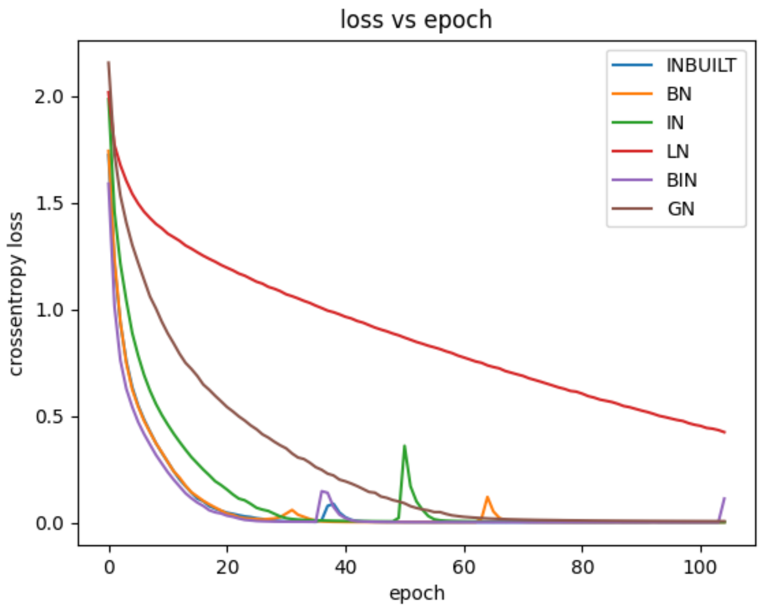


Figure 9: Train loss for different normalization implemented

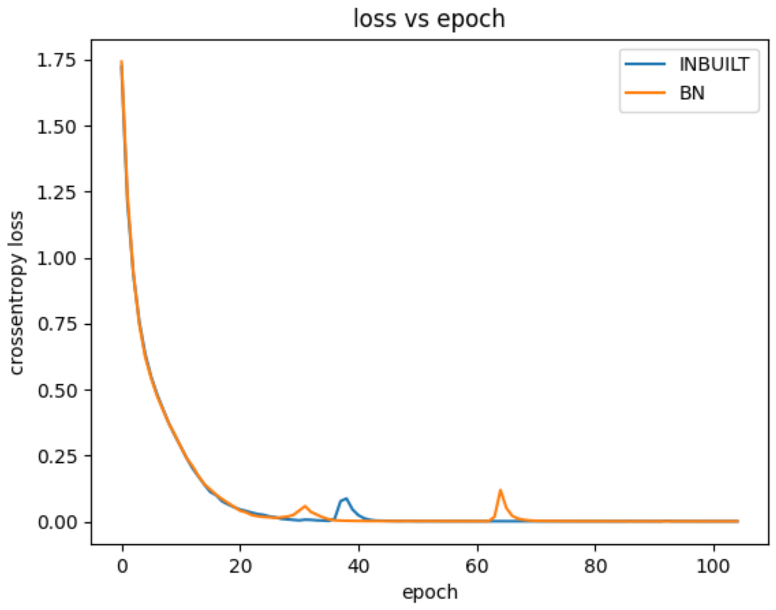
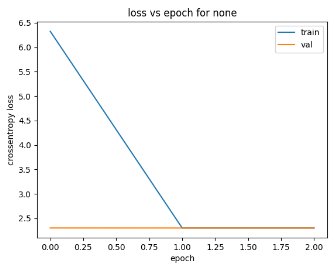
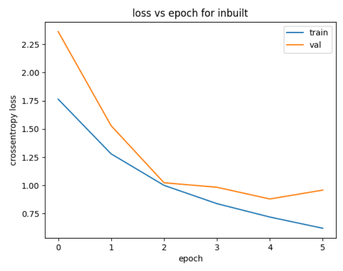


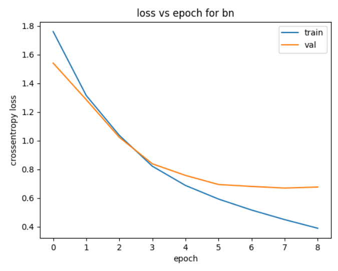
Figure 10: Train loss for inbuilt and implemented batch normalization



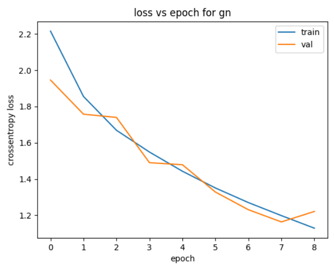
(a) None



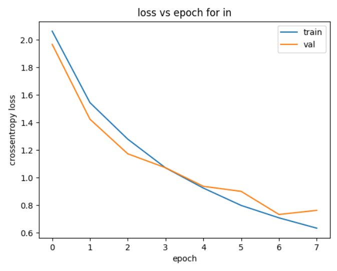
(b) Inbuilt



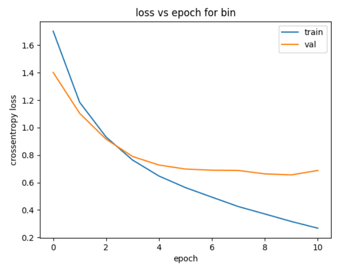
(c) BN



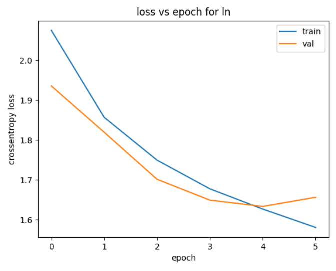
(d) GN



(e) IN

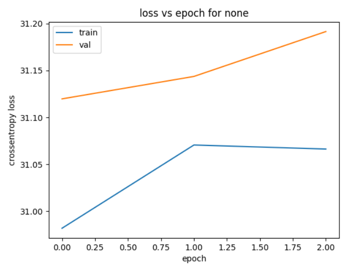


(f) BIN

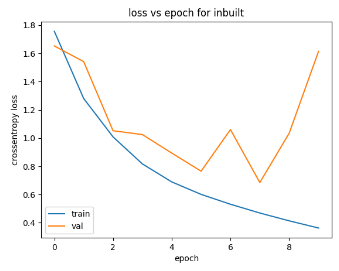


(g) LN

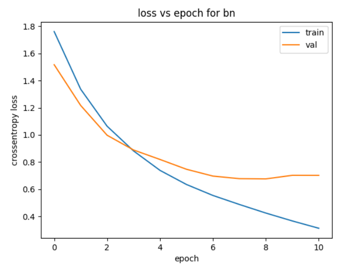
Figure 11: Train and val loss for different normalizations at $patience = 1$



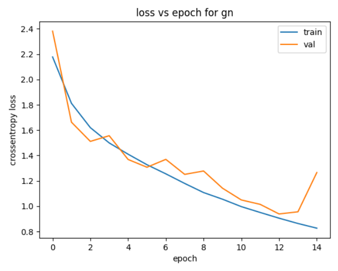
(a) None



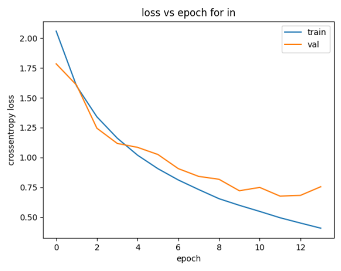
(b) Inbuilt



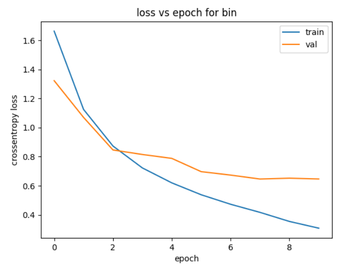
(c) BN



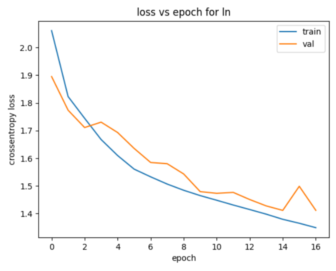
(d) GN



(e) IN

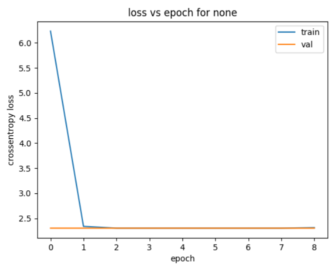


(f) BIN

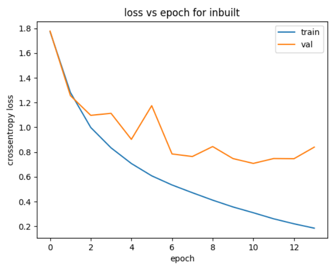


(g) LN

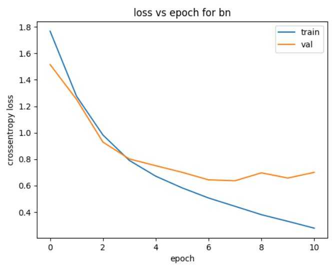
Figure 12: Train and val loss for different normalizations at $patience = 2$



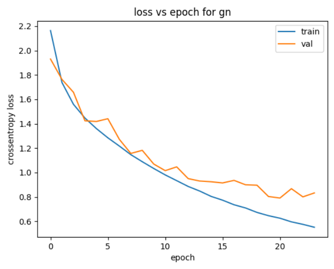
(a) None



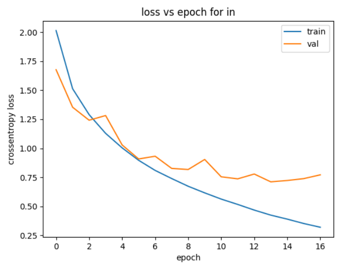
(b) Inbuilt



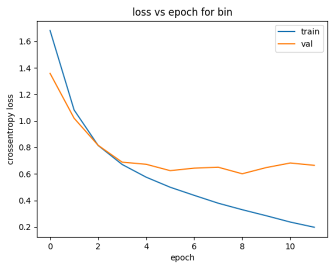
(c) BN



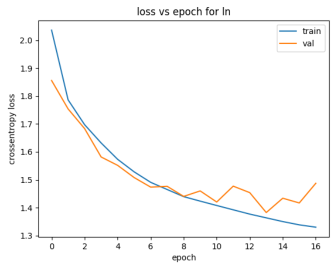
(d) GN



(e) IN

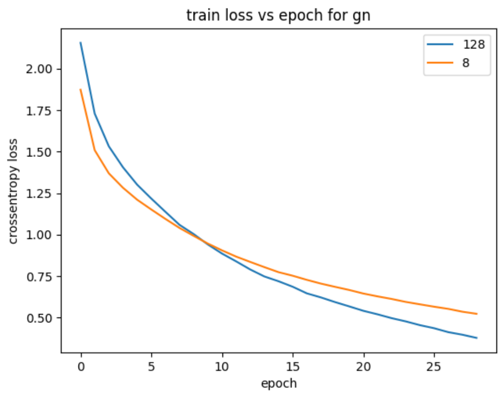


(f) BIN

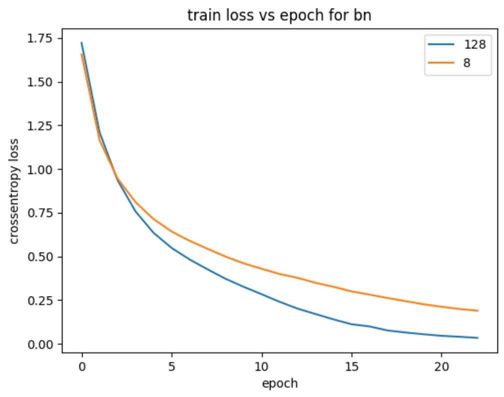


(g) LN

Figure 13: Train and val loss for different normalizations at $patience = 3$

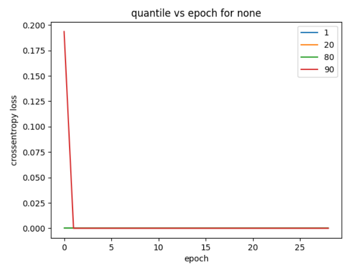


(a) BIN

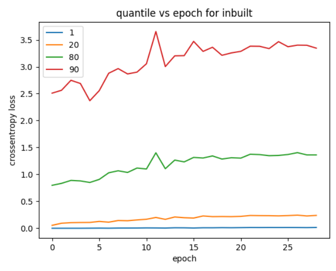


(b) BN

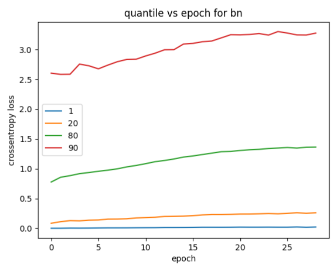
Figure 14: Training loss for different batch sizes



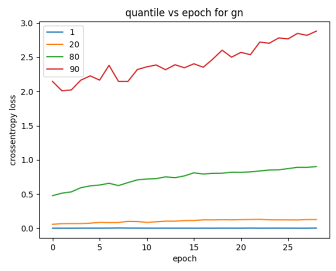
(a) None



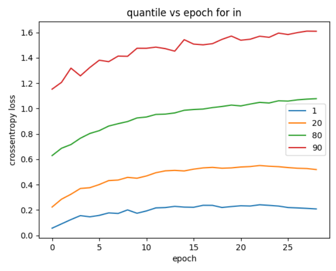
(b) Inbuilt



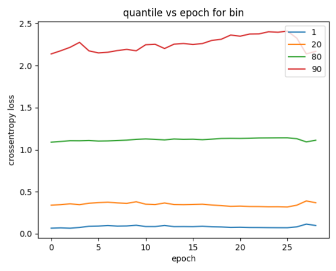
(c) BN



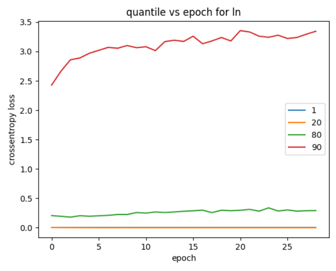
(d) GN



(e) IN



(f) BIN



(g) LN

Figure 15: Train and val loss for different normalizations at $patience = 3$

3 Text to SQL

Approach

This problem is similar to machine translation but has few differences from the vanilla machine translation problem. The first difference is that now translation is also dependent on the *database* for which the text is given. For example for text *How man employees are there* ? will have completely different *SQL* query for database *institute* and *corporate*. There is some more syntactic structure to *SQL* from the plain natural language which helps us to exploit this idea to make a better model.

Structure Exploitation

As *SQL* query has *primary keys* and *values* as one of its parts which can be any possible text. This would mean that the decoder vocabulary should incorporate this. But we can exploit one idea that most of *primary keys* and *values* are tokens in the input sequence and we decoder just needs to learn to copy and paste it without worrying about its possible values. To exploit this property I ran regular expression checks and preprocess the training data. For example:

```
    Training sample: [(How many cities have area greater than 2000 acers but population less
than 10000), SELECT name FROM cities WHERE area > 2000 AND population < 10000]
```

Parser will parse the input sentence s and generate value token mappings

```
    text_preprocess(s) = [(How many cities have area greater than VALUE_0 acers but population
less than VALUE_1),{2000: VALUE_0, 10000: VALUE_1}]
```

And preproceesing SQL query q with value mapping m gives:

```
    sql_preprocess(q,m) = [SELECT name FROM cities WHERE area > VALUE_0 AND population < VALUE_1]
```

The regular expression that I used are

```
Dates:
[\"\']?(\d+[-./|]\d+[-./|]\d+)[\"\']?
[\"\']?(\d+[-./|]\d+)[\"\']?

Time:
[\"\']?(\d+[-./|:]\d+[-./|:]\d+(?:|am|pm))[\"\']?
[\"\']?(\d+[-./|:]\d+(?:|am|pm))[\"\']?

Number:
[\"\']?(\d+(?:|th|nd))[\"\'.]?

Word in the quotation:
\" ?([\w+ ?]+) ?\"
\' ?([\w+ ?]+) ?\'
\' ?([\w. ?]+\w*) ?\'
```

Different Databases

To incorporate different databases in the output probabilities I added a database-specific mask that masked non-relevant tokens like column names, and table names from other databases.

Models

LSTM-LSTM

The model was trained with a single lstm layer encoder. And hidden size were 128 dimensions of both the encoder and decoder lstm. Teacher forcing was used during training with a probability of 0.5. Decoder vocabulary was having 50 dimensional embeddings. Following hyperparameter search was tried

- Bilstm vs Lstm encoder didn't affect the model much. But surprisingly Lstm encoder was giving more accuracy than Bilstm
- Different teacher forcing probabilities were tried. For higher teachers forcing the probability model didn't give good val accuracy and for lower it was not training. Optimal was found to be 0.5. THis was expected as exposure bias kicks in at higher teacher-forcing probabilities.
- Hidden dimensions of the LSTM were tried but 128 gave the best representational capacity and trainability. This was also expected due to overfitting.
- Beam search k was found to play role in the val accuracies. For k ¿= 10 I got best results. This was again expected as for larger k we are actually doing whole search.

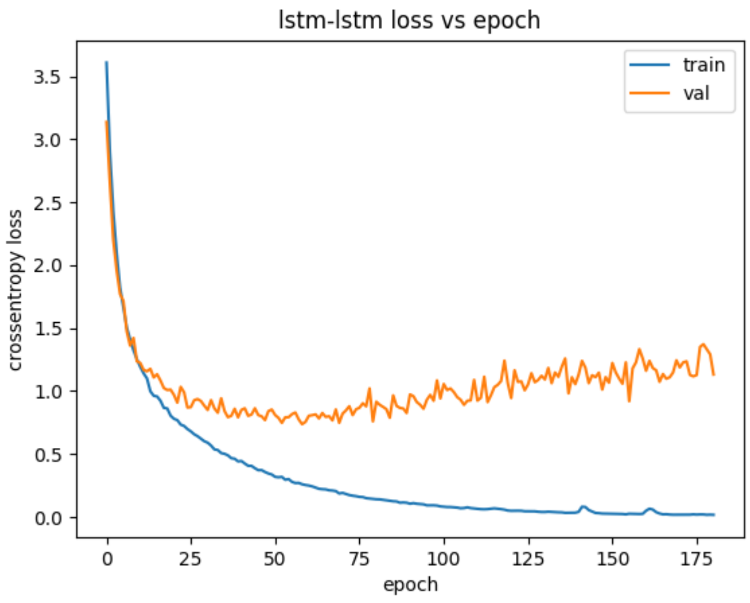


Figure 16: Lstm-lstm loss curves

LSTM-LSTM With attention

The model was trained with a single lstm layer encoder. And hidden size were 128 dimensions of both the encoder and decoder lstm. The hidden size in attention was also 128 dimensions.

- Bilstm vs Lstm same as above
- Different teacher forcing probabilities same as above
- Hidden dimensions of the LSTM were tried but 128 gave the best representational capacity and trainability. This was also expected due to overfitting.
- Beam search k same as above

- Hidden attention dimension was found to be optimal at 128. 256 and 64 were tried but both gave worse result at same epoch 25.

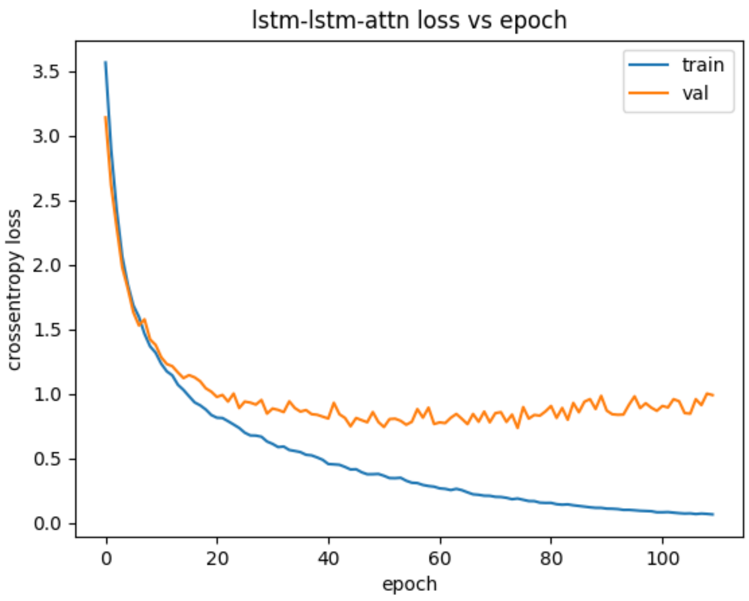


Figure 17: Lstm-lstm attention loss curves

BERT Frozen-LSTM With attention

Bert-base-cased was used as the encoder. The hidden dimension of bert is 768 aprt from that all the hyperparameters are same as above.

- Different teacher forcing probabilities same as above
- Hidden dimensions of the LSTM were tried but 128 gave the best representational capacity and trainability. This was also expected due to overfitting.
- Beam search k same as above
- Hidden attention dimension was found to be optimal at 128. 256 and 64 were tried but both gave worse results at the same epoch 25.

BERT-LSTM With attention

The only change here was to set the required grad for Bert as True. Even though the model capacity was high but due to high lr of encoder bert the model performed very bad. I adjusted the lr of the encoder as 0.05 times the true lr. It gave greate results.

Key Insights

- Initializing decoder vocabulary with glove embedding improves the results. Even though the SQL keywords like SELECT are different from the English select still the high-level meaning are same.
- Reducing the learning rate of the Bert encoder improves the model's overall performance. This guides the model that the encoder doen't need big updates but the decoder does.

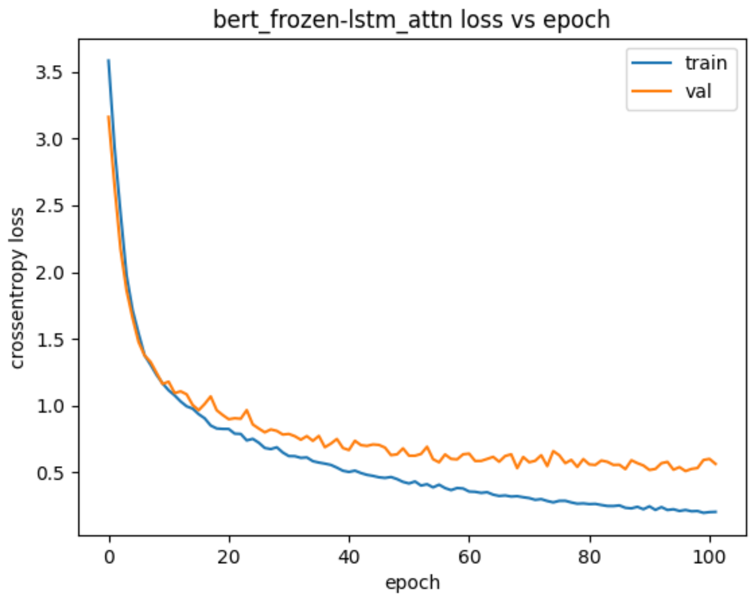


Figure 18: Bert-Frozen-lstm attention loss curves

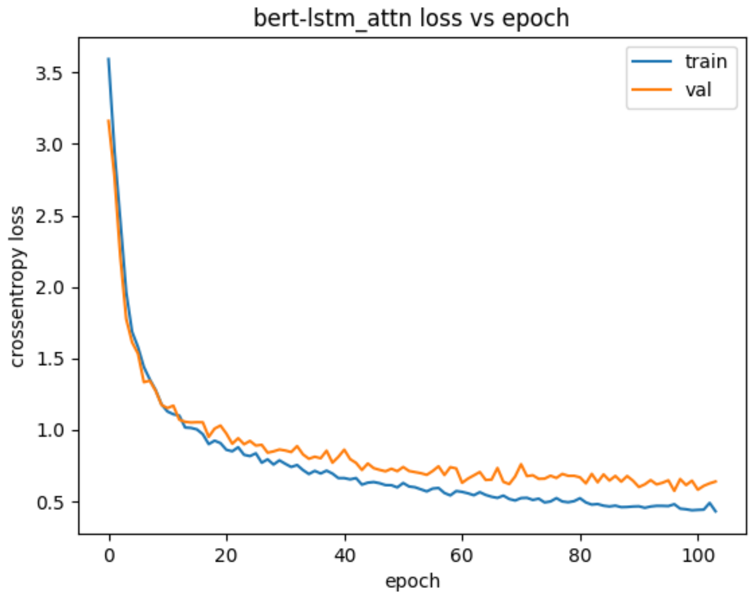


Figure 19: Bert-lstm attention loss curves

- Bislstm and lstm perform equally well as an encoder in this case.
- Increasing the hidden dimensions of the LSTM cell does not help as due to the curse of dimensionality.

Model	Easy	Medium	Hard	Extra	All
LSTM-LSTM	0.264	0.075	0.092	0.011	0.109
LSTM-LSTM-Attn	0.286	0.075	0.153	0.019	0.129
BERT-frozen-LSTM-Attn	0.407	0.146	0.186	0.027	0.191
BERT-LSTM-Attn	0.386	0.126	0.146	0.011	0.168

Table 3: Exact match accuary

Model	Easy	Medium	Hard	Extra	All
LSTM-LSTM	0.273	0.084	0.130	0.027	0.126
LSTM-LSTM-Attn	0.320	0.084	0.182	0.030	0.149
BERT-frozen-LSTM-Attn	0.400	0.157	0.231	0.057	0.209
BERT-LSTM-Attn	0.398	0.137	0.205	0.046	0.194

Table 4: Execution accuary