COL775: Deep Learning

Assignment 1.2: Text-to-SQL: A challenging Sequence to Sequence task

Student Name: Aman Bhardwaj **Entry Number:** 2022ANZ8400

MODELS URL:https://drive.google.com/drive/folders/1TDidPlvSPQ7YnYBgn2A56rWgy2AZgkR-?usp=sharing

Neural Machine Translation (NMT) is an active area of research in the field of NLP. Initially it started with automated **natural language (NL)** translation between different languages. Same translation engines were extended to translation between **programming languages (PL)** using language models for PL. Some of the tasks fall under the category of NMT are: • NL-to-NL • NL summarization • NL-to-PL • PL-to-NL • PL summarization. **Text-to-SQL task fall under the category of NL-to-PL NMT.**

1 Implementation Details

In this assignment we experiment with 4 different model architectures for text-to-SQL conversion.

1.1 Pre-processing and Vocab Building:

It involves below parts:

1. **Input sanitizing:** This includes pre-processing of both NL and SQL text such as conversion of symbols like ” $<=$ ” $\rightarrow$ ” $\leq$ ”, ”T4.column_name” $\rightarrow$ ”T4” ”.” ”column_name” etc.
2. **Tokenizer:** For questions (NL) we use tokenier for english from `spacy` library. We further built a custom tokenizer for SQL queries.
3. **Encoder Vocab:** Encoder vocab was built using **tokens from questions**. **Minimum frequency** for a token to be included in the vocab was kept to be 3.
4. **Decoder Vocab:** Decoder vocab includes **tokens from SQL queries and from the Table Schemas** provided in `tables.json`.
5. **Bert-tokenizer:** For bert based encoders, we use `bert-tokenizer` from transformer library from huggingface. Therefore, we use the corresponding vocab as well.

1.2 Seq2Seq Model with GloVe Embeddings: LSTM Encoder and LSTM Decoder

This Seq2Seq architecture has been developed as per following details:

1. **Bi-directional LSTM Encoder:**
   - We use default implementation from pytorch for bi-LSTM cell (`nn.LSTM`).
   - **Dropouts:** Between multiple LSTM cells we use dropouts with probability $p = 0.3$.

- In case of bi-directional lstm we need to match the dimensions of hidden and cell units with decoder input. For that we use `nn.Linear`.

2. **Uni-directional LSTM Decoder:** `nn.LSTM` was also used for the implementation of decoder.
3. **GloVe Embeddings:** We use pretrained glove embeddings from `torchtext` library to initialize the encoder embeddings.
4. **Optimizer and Scheduler:** We use Adam optimizer and CosineAnnealingLR scheduler.
5. **Loss:** We use CrossEntropy loss for this task.

For the experimentation part, we experiment with different number of LSTM layers in both encoder and decoder, different embedding size, different number of hidden units in both encoder and decoder.

1.3 Seq2Seq + Attention Model with GloVe Embeddings: LSTM Encoder and LSTM Attention Decoder

In addition to the previous implementation, here we modify the decoder part to Attention Decoder. We implement **AttentionNetwork** class using `nn.Module` and integrate it with the decoder.

Please note, we had to modify the decoder in order to accomodate the context from the attention network at the input dimension of decoder LSTM cell.

1.4 BERT Seq2Seq + Attention Model with frozen BERT-base-cased: BERT Encoder and LSTM Attention Decoder

Here new addition is BERT encoder. Details are as follows:

1. **BERT-base-cased:** In order to implement BERT based encoder we use the BERT base implementation from **huggingface's transformer** library and used the pre-trained weights available by them.
2. **Freeze fine-tuning:** In order to freeze the fine-tuning of bert model imported, we use `require_grad = False` for all the parameters of the model.
3. **Encoder output:** We use `last_hidden_state` of the output as the bert-encoder-output.

For this model we experiment with different learning rates. `lr=0.0001` seems to be working best for this model.

1.5 BERT Seq2Seq + Attention Model with fine-tuned BERT-base-cased: BERT Encoder (tunable) and LSTM Attention Decoder

There are a total of **12-layers** in BERT. For this part, we experiment with following three configurations.

- All 12 layers open for fine-tuning.
- Last 4 layers open for fine-tuning.
- Last 8 layers open for fine-tuning.

2 Comparison of Loss Curves

Figure 1 illustrates the comparison of train and validation loss values for 200 epochs illustrated on the same graph for all the 4 architectures. The loss values used for this plots are from the best performing models from all the 4 models.

Furthermore, in order to compare the train loss curves and val loss curves with other models, we plot them on two separate graphs in Figure 2.

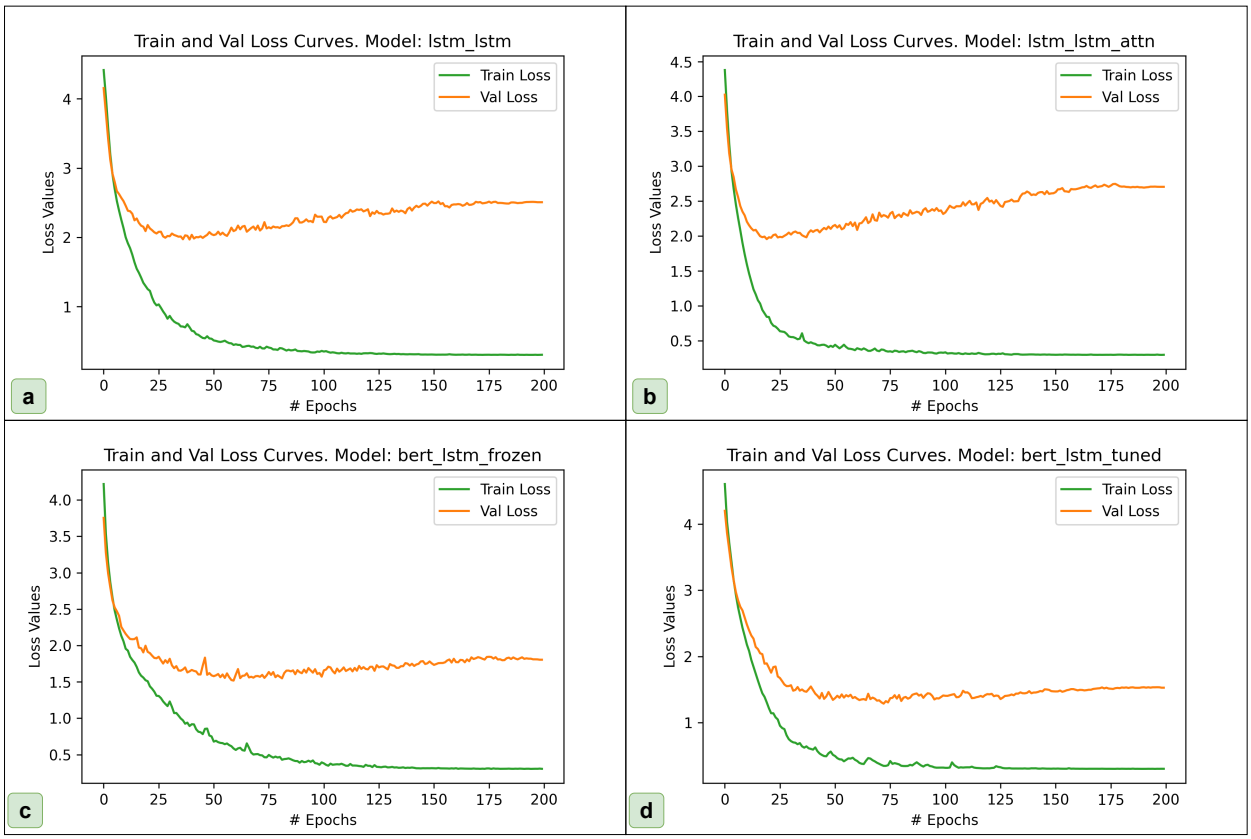

Figure 1: Train and Loss curves of best performing individual models (a) `lstm_lstm`, (b) `lstm_lstm_attn`, (c) `bert_lstm_attn_frozen`, (d) `bert_lstm_attn_tuned`

Observations and inferences from loss curves in Figure 1

1. **Train Loss:** For all four curves in Figure 1 train loss reduces continuously with increase in epochs and settles to a low value close to 0.4 for all the cases. Initially it shows small irregularities, but after 75 epochs it becomes smooth.

2. **Validation Loss:** For all four models, val loss reduces initially till 30-40 epochs, then it increase a bit and then stays almost smooth towards the end of training. This shows model settles in a local minima valley.

3. **Final val loss values:** Final val loss values are in the order of $lstm\_lstm > lstm\_lstm\_attn > bert\_lstm\_attn\_frozen > bert\_lstm\_attn\_tuned$. This shows that the model performance should be in reverse order. We will validate if this is true in the next sub-section.

Observations and inferences from loss curves in Figure 2 In order to compare the all four models together in terms of loss values, we plan to analyse Figure 2 below:

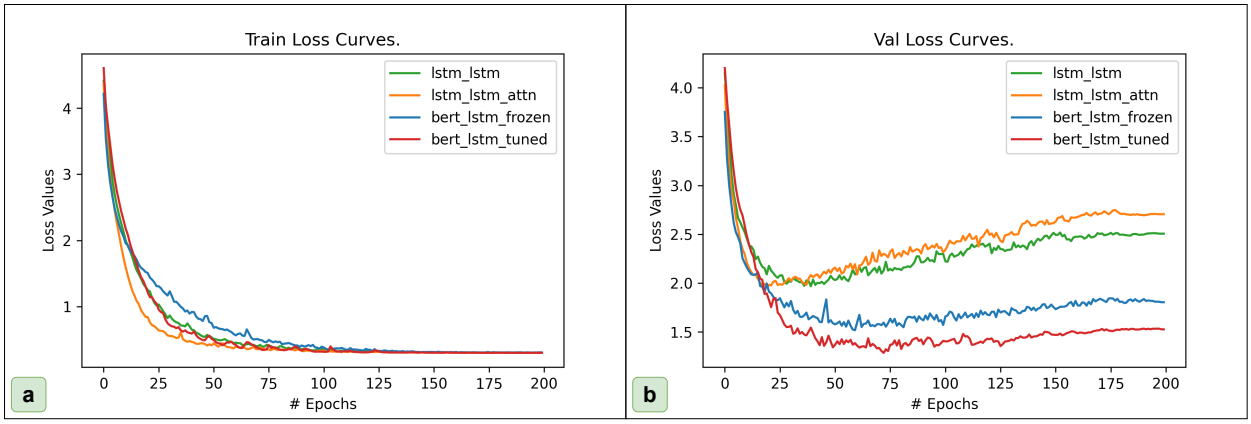

Figure 2: (a) Train and (b) Loss curves compared for all the models

1. **Train Loss for all four models:**
   - **Initial and Final Values:** Figure 2 (a) shows that the initial values of all four models are close to 4.5 and final values are also same close to 0.4.
   - **Rate of decay:** Initial rate of decay is fastest for $lstm\_lstm\_attn$ and slowest for $bert\_lstm\_attn\_frozen$. Rest two models have almost similar rate of decay for train loss.
2. **Val Loss for all four models:**
   - **Initial rate of decay:** Initial rate of decay is fastest for $lstm\_lstm\_attn$ and $bert\_lstm\_attn\_frozen$ and slowest for $bert\_lstm\_attn\_tuned$.
   - **Increase in loss values:** After initial decay, the loss values increase for all the models. However, increase is maximum in $lstm\_lstm\_attn$ and minimum in $bert\_lstm\_attn\_tuned$. For $bert\_lstm\_attn\_tuned$, the increase is almost negligible. This shows that of all the four models, learning is better for $bert\_lstm\_attn\_tuned$.
   - **Final values of val loss:** Final val loss values are in the order of $lstm\_lstm > lstm\_lstm\_attn > bert\_lstm\_attn\_frozen > bert\_lstm\_attn\_tuned$. This shows that the model performance should be in reverse order. We will validate if this is true in the next sub-section.

3 Comparison of Execution and Exact Match Accuracy

Table 1 contains the performance metrics for all the models along with their best performing parameter configurations.

Observations and inferences from performance metrics Table 1

1. **Model performance order:** For the hyper-parameter configuration reported in Table 1 the order of **Execution Accuracy (EA)** is $lstm\_lstm < lstm\_lstm\_attn = bert\_lstm\_attn\_frozen << bert\_lstm\_attn\_tuned$ while the **Exact Match Accuracy (EMA)** order is $lstm\_lstm < lstm\_lstm\_attn < bert\_lstm\_attn\_frozen << bert\_lstm\_attn\_tuned$. Therefore, we can clearly conclude that the performance of $bert\_lstm\_attn\_tuned$ is best (EA = 29.6% and EMA = 33.7%), whereas, worse performing model is $lstm\_lstm$ (EA = 21% and EMA = 22.2%).
2. Above results are in consensus with val loss values discussed in previous subsection.

Table 1: Execution and Exact Match Accuracy for best performance model of all 4 configurations.

Model	Accuracy (%)		Hidden Size		Embedding Dim		Batch	Epochs	LR
	Execution	Exact Match	Encoder	Decoder	Encoder	Decoder			
lstm_lstm	21%	22.2%	512	512	300	300	32	200	0.001
lstm_lstm	21.4%	22.9%	512	512	300	300	16	300	0.001
lstm_lstm_attn	23.3%	24.7%	512	512	300	300	32	200	0.001
bert_lstm_frozen	23.3%	25.1%	-	768	768	300	32	200	0.0001
bert_lstm_tuned (Last 4 Layers)	28.9%	32.3%	-	768	768	300	32	200	0.0001
bert_lstm_tuned (Last 8 Layers)	**29.6%**	**33.7%**	-	768	768	300	32	200	0.0001

Note: Number of LSTM layers = 1 (where ever applicable) for all the above models

3. **Intuitive Sense:** The performance results make intuitive sense as per the sophistication of models as BERT based encoder are better than LSTM based encoders and hence better performing.

4. **Comparison of bert_lstm_attn_tuned with rest:** The performance values of rest of the three models are quite close (approx difference of 1-1.5%), however, *bert_lstm_attn_tuned* **outperforms** the best of the rest three models **by a huge margin** of 6.3% in EA and 8.6% in EMA with a configuration of last 4 layers tunable.

4 Hyper-parameter settings used for experimentation

Below are the hyper-parameter settings that have been kept as arguments for the experimentation:

1. **MODEL_TYPE** Select the model you want to run from [lstm_lstm, lstm_lstm_attn, bert_lstm_attn_frozen, bert_lstm_attn_tuned]

2. **NUM_WORKERS** Number of workers used for dataloading.

3. **BATCH_SIZE** Batch size to be used during training.

4. **EPOCHS** Max number of epochs for model to train

5. **EN_HIDDEN** Encoder Hidden Units

6. **DE_HIDDEN** Decoder Hidden Units

7. **EN_NUM_LAYERS** Number of lstm layers in encoder.

8. **DE_NUM_LAYERS** Number of lstm layers in decoder.

9. **EMBED_DIM** Embeddings dimension for both encoder and decoder.

10. **DEVICE** Device to run on.

11. **RESTORE** Restore last saved model.

12. **SEARCH_TYPE** Type of greedy, beam search to perform on decoder.

13. **BEAM_SIZE** Beam size to be used during beam search decoding.

14. **BERT_TUNE_LAYERS** Number of last layers to finetune for bert encoder.

15. **LR** Learning rate for the model

Note: All the models have been developed with generic code that builds over above parameters that can be used to configure encoder and decoder during model training as python arguments.

5 Hyper-parameter tuning and grid search

Table 2: Details parameter tuning/grid search performed

S. No.	Accuracy (%)		# LSTM Layers		Hidden Size		Embedding Dim		Batch	Epochs	LR
	Execution	Exact Match	Encoder	Decoder	Encoder	Decoder	Encoder	Decoder			
lstm_lstm											
1	**21%**	**22.2%**	1	1	512	512	300	300	32	200	0.001
2	19.4%	20.9%	1	1	512	512	300	300	32	300	0.001
3	**21.4%**	**22.9%**	1	1	512	512	300	300	16	300	0.001
4	17.7%	18.4%	1	1	256	256	300	300	32	200	0.001
5	16.3%	17.5%	2	2	256	256	300	300	32	200	0.001
6	13.7%	14.1%	2	2	1024	1024	300	300	32	200	0.001
7	4.5%	2.5%	4	4	512	512	300	300	32	200	0.001
8	0**	0**	4	4	1024	1024	300	300	32	200	0.001
9	6.9%	5.4%	1	1	512	512	300	300	32	200	0.0001
lstm_lstm_attn											
1	**23.3%**	**24.7%**	NA	1	512	512	768	300	32	200	0.001
2	0.001 †	0.0 †	NA	1	512	512	768	300	32	200	0.0001
bert_lstm_attn_frozen											
1	**23.3%**	**25.1%**	NA	1	-	768	768	300	32	200	0.0001
bert_lstm_attn_tuned (Tunable Layers = (1) All, (2) last 4, (3) Last 8 respectively)											
1	7.9%	8.2%	NA	1	-	768	768	300	32	200	0.0001
2	28.9%	32.3%	NA	1	-	768	768	300	32	200	0.0001
3	**29.6%**	**33.7%**	NA	1	-	768	768	300	32	200	0.0001

**Model not able to train till 109 Epochs. † Accuracy achieved till 20th Epoch

Observations and inferences from Table 2

All the configurations of model hyper-parameter use for experimentation for have been reported in Table 2 along with respective performance numbers.

1. **lstm_lstm:** For this model we experiment with different number of LSTM layers in encoder and decoder (1/2/4), number of hidden units in LSTM cell (256/512/1024), learning rates (0.001/0.0001) and Epoch Size (16/32).

    - **Smaller batch size** can fetch you better performance as number of padded tokens after shorter sentences reduces. But you will have to increase total epochs in order to achieve the performance gain.
    - More number of LSTM layers in encoder and decoder reduces model performance.
    - 512 is the optimal hidden units number in LSTM encoder and decoder as the performance reduces for 256 and 1024.
    - Decreasing learning rate to 0.0001 increases learning time and reduces performance.

- Training LSTM with higher number of hidden layers becomes difficult and extremely slow.
- Best configuration for this model has been highlighted in the Table.

2. **lstm_lstm_attn:**
   - LSTM encoder with attention decoder shows improvement in results at the same configuration that performs best for previous model.
   - LR=0.0001 is quite small LR for this model as the model is able to achieve 0.001 EA in 109 epochs.
3. **bert_lstm_attn_frozen:** To enable good training for this model we have to experiment mainly with learning rate. Therefore, we experiment with LR=0.01, 0.001, 0.0001, and 0.00001. The LR=0.0001 seems to be working best for this model as well.
4. **bert_lstm_attn_tuned:**
   - For this model we use the same configuration used for the previous model, however, we experiment with number of layers of BERT encoder to open up for training.
   - **All layers fine-tuning:** Enabling gradients for all the layers is a bad idea. Firstly, it increases the training time by 3 times and secondly, it messes up with the pre-trained model and can take model far from the learned local minima.
   - **Fine-tuning Last few layers:** We experiment with tuning of last 4 and 8 layers. 8 layer tuning performs the best and it outperforms other model by a huge margin.

6 Insights gained from the training procedure and the outputs of models

1. **Model complexity and performance** Increasing model complexity doesn't necessarily increase model performance, as it can be seen that with increase of number of LSTM layers to 2 and 4 in Encoder and Decoder decreases the performance and makes model slow.
2. **Learning Rate:** For sequential models LR is highly critical as the performance of these models are highly prone changes in learning rate.
3. **BERT Fine-tuning:** Opening up all the layers for fine-tuning completely messes-up with the originally trained model. Hence, open-up last few layers for fine-tuning, or add up a few new Linear layers and train them.
4. **Neural Machine Translation for programming languages:** requires huge dataset in order to work efficiently without using domain knowledge in model designing and training. It can be tricky and a lot more difficult than NL translation, as each PL has to be syntactically correct in order to work unlike NL which can work even if there are some minor mistakes.
5. **Regular Git commits:** Do regular code commits of your code especially the working version before adding up new functionality to it. Otherwise you might have to spend a whole day to debugging what went wrong. Lol... :D