Assignment 4 - Shortcut-Stacked Sentence Encoders for Multi-Domain Inference

Otmazgin, Shon 305394975

Rassin, Royi 311334734

Submitting Username: noyiras

1 Why this paper?

While perusing through the papers, we sought a work that is interesting, well-written, without any details omitted. Since we already delved quite in-depth into LSTMs in assignment two and three, and are fascinated by them, a paper such as this was the obvious choice. The other reason we found this paper particularly appealing, is its surprising simplicity in solving this task.

2 What is their method?

The model is made of two parts. The encoder(s): a stacked bidirectional-LSTM with shortcut connections, followed by a max-pooling layer. And an entailment classifier.

The stacked encoders encode the premise and the hypothesis (and in order to fully-generalize, the same encoders are applied to both) to vectors, and the classifier uses the vectors to label the relation between the premise, and the hypothesis (this is the task we are solving); entailment, contradiction, or neural.

In a 'shortcut connections' setting, the input sequence for the ith biLSTM layer is the word-embedding, followed by the concatenated outputs of **all the previous layers**. This idea relates to residual connections in CNNs, which is widely applied in the field of Computer Vision. The proposed encoder does not require any syntactic information, attention or memory structure. After receiving the encoded premise and hypothesis, three matching methods are applied by the classifier: concatenation, element-wise distance, element-wise product, and their match-vectors are concatenated. Mathematically:

$$m = [v_p, v_h, |v_p - v_h|, v_p * v_h]$$

Finally, the concatenated result m is fed into an MLP-layer with a softmax layer, to make the classification. In terms of hyper-parameters, they opted for a batch-size of 32, a cross-entropy loss with an ADAM based optimizer where

the initial learning-rate is 0.0002 with half decay every two epochs. Further, the number of hidden units for the MLP is 1600 and a Dropout layer right after it with probability set to 0.1. They only experimented with up to 3-layers with 512, 1024, 2048 units respectively. They used pre-trained 300D GloVe 840B vectors as their embeddings.

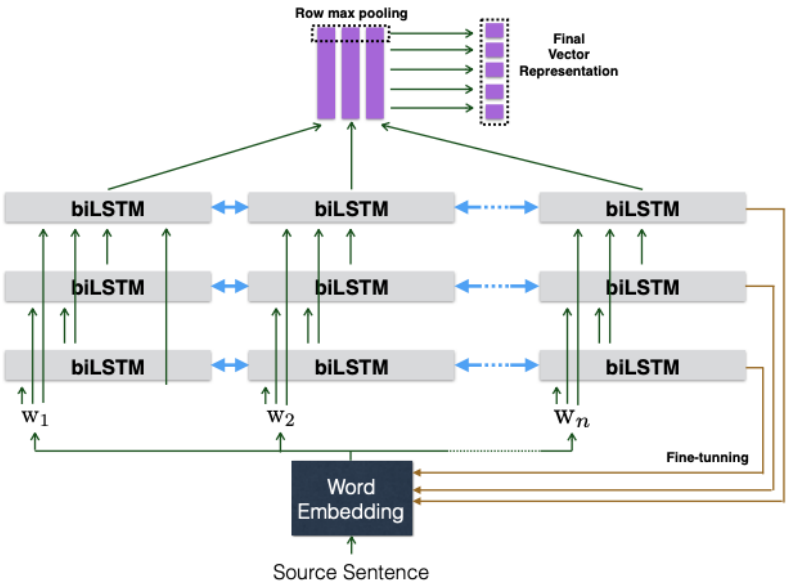

Figure 1: Architecture

3 Results

3.1 Their results and modifications made to follow the SNLI leaderboard

To comply with the SNLI leaderboard settings, the original architecture had to be modified to one with less parameters. Instead of applying shortcut connections, they apply residual ones: the input to each next BiLSTM layer is the concatenation of the word embedding and the summation of the outputs of all previous layers. By applying this change, the number of parameters was significantly reduced (to 9.7M). Additionally, the MLP-layer was of 800 units instead of 1600. The authors report 85.7 accuracy with this architecture.

Model	#param	Dev	Test
300D Residual-Stacked-Encoder	9.7M	86.4	85.7
600D Residual-Stacked-Encoder	28.9M	**87.0**	**86.0**
600D Shortcut-Stacked-Encoder	34.7M	86.8	85.9

Figure 2: Residual Networks

3.2 Our results

We trained the network with three types of embedding-sets: 8B, 42B (the authors did not train smaller models like we did), and 840B (the one they submitted to the leaderboard). Also, by introducing AdamW, SpaCy's tokenizer, and assigned UNK to be the average vector of the entire corpus, we slightly outperformed them (our 840B version reached 86.1 accuracy). It is crucial to emphasize how surprising it is that the 8B model is so near in accuracy (just 0.8 less accurate!) to the 840B version, and **is significantly** less demanding, and faster to train in comparison to the 840B version that required renting an Azure server and was slow to train.

Embedding-Set	Test Acc
8B	85.3
42B	85.7
840B	86.1

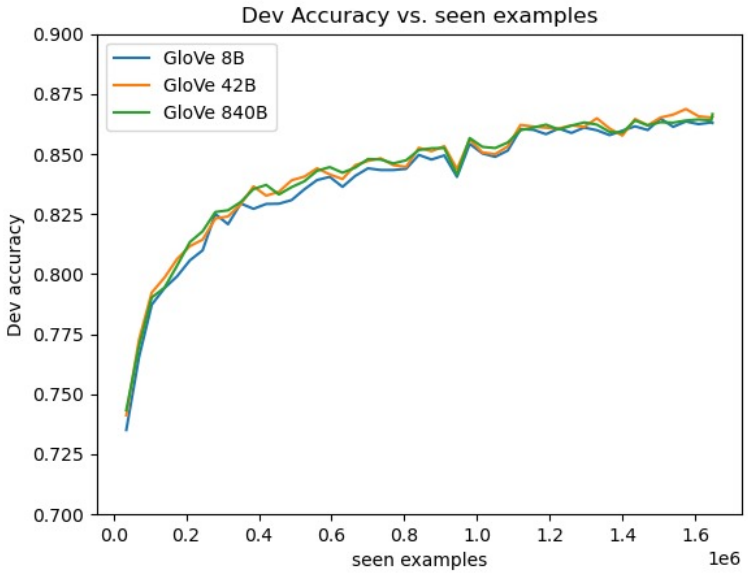

Figure 3: Dev Accuracy

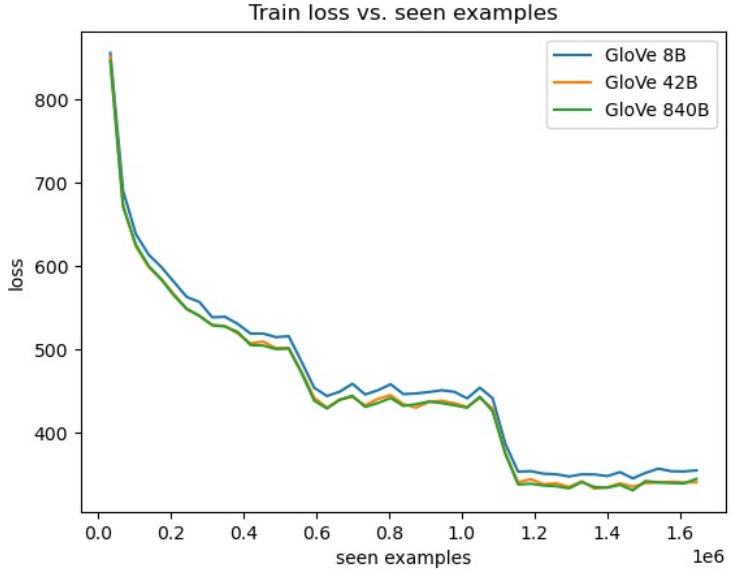

Figure 4: Train Loss

3.3 Replicating results

3.3.1 Challenges

The biggest obstacle we had to deal with was actually running the network: the embedding-set they used was highly demanding of our systems - completely depleting our RAM, and when we tried to run it on a Google CoLab Pro version, the CUDA memory was depleted. Encountering RAM and CUDA-memory issues led us to training the model in smaller settings; 8B and 42B, which the authors did not do. Eventually we rented some time on an Azure system to run the large model. Additionally, it was unclear what tokenizer (or even a rough idea) was used, and how the authors dealt with unknown words.

3.3.2 What worked straightforward

Other than the challenges we mentioned above, the encoders were straightforward, mainly because the authors mentioned most of the important details that are required to replicate their work, had figures in the paper, that their writing-flow was easy to follow, and an attached implementation in PyTorch.

4 Conclusion

Completing this project improved our confidence in reading and replicating papers, as well as our ability to deal with unexpected issues; this is the first time we faced with a model that Google CoLab Pro is not capable of running. Further, dealing with missing information, such as the way they address unknown words, and what tokenizer they used, led us to come up with our own ideas, which eventually led to an improvement in the model's performance! The next time we will read through a paper, we will not be deterred by implementing their work.