STEVE GUTFREUND – 342873791
YOEL JASNER – 204380992

The last 2+ weeks were very exhausting. We spent hours on this exercise. We tried more than one paper, because sadly enough we failed them, as will be explained, but at least we learned some new stuff and got to understand some topics more profoundly.

So, as we have put very much effort in this exercise, we'll briefly go over the 3 papers we tried

The paper(s)

1. Learning Natural Language Inference using Bidirectional LSTM model and Inner-Attention by Yang Liu et al. '16 with an accuracy of 84.2%.
   The reason of interest in this paper is the logic of their model. Their model is based on sentence encoding with a BiLSTM layer and a mean-pooling layer above that. The most exciting part is the next stage, they use a self-attention mechanism in which they give a higher score to more important words in the sentences. This is a very human-like reasoning when comparing 2 sentences. We very excited to see a machine which really 'thinks' like a human-being. After encoding the 2 sentences they created one 'big' vector (similar to what we did in our final model) and fed that to a softmax layer to get the classification probabilities.
   Following is a nice diagram they created for that:

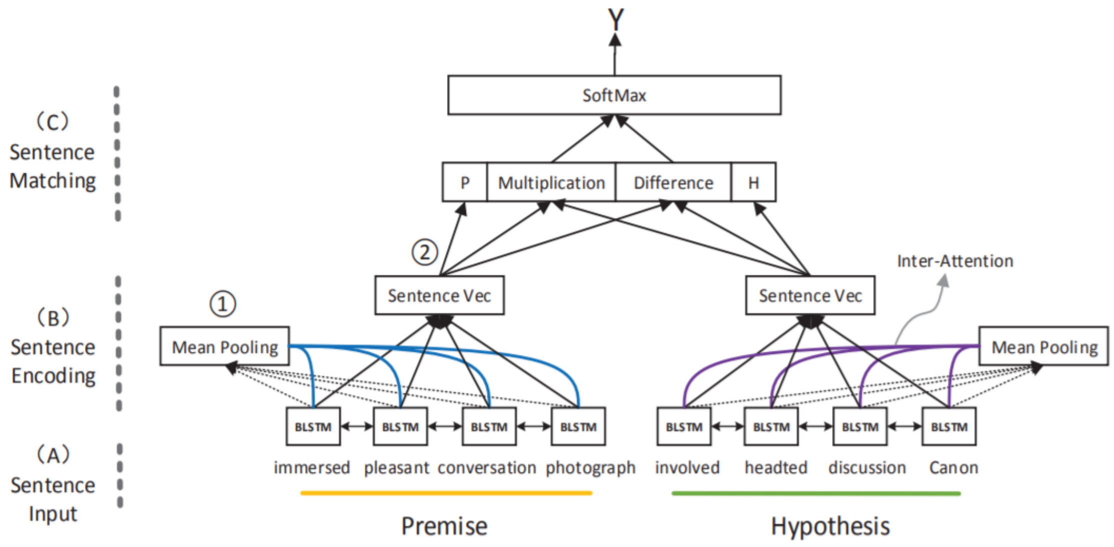
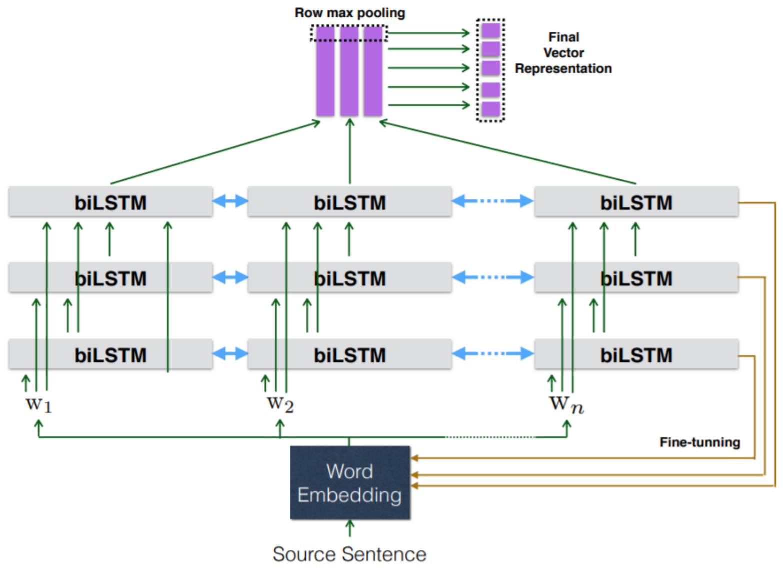


Figure 1: Architecture of Bidirectional LSTM model with Inner-Attention

   The problem with this paper was the implementation.
   The BiLSTM and the pooling layer should not have been a problem. We were concerned about the self-attention. They do not give enough information for us to understand how it should be implemented.
   We started reading about attention mechanisms, but time started slipping away...
   So, we decided to change paper ☹

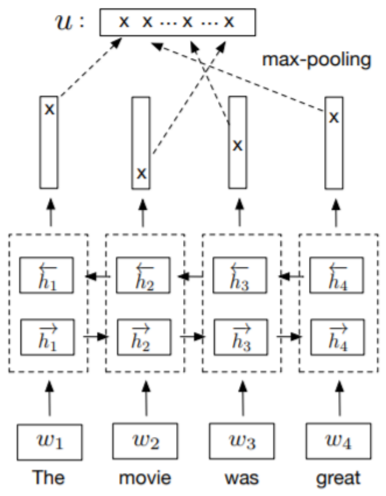
2. Shortcut-Stacked Sentence Encoders for Multi-Domain Inference by Nie and Bansal ’17 with an accuracy of 85.7%
   The reason for this one, was its simplicity (in terms of implementation, at least that's what we thought).
   We can divide the model in two stages (this is right for most of the papers):
   - sentence encoder (producing a vector representation for a sentence):
     This is done by first using word embeddings and then the sentence is fed into 3 BiLSTM layers. The input to the i-th BiLSTM is a concatenation of the very first inputs and the outputs of the first i-1 BiLSTMs.
     This results in a matrix on which we perform row-max pooling to finally receive a vector representor for the sentence.



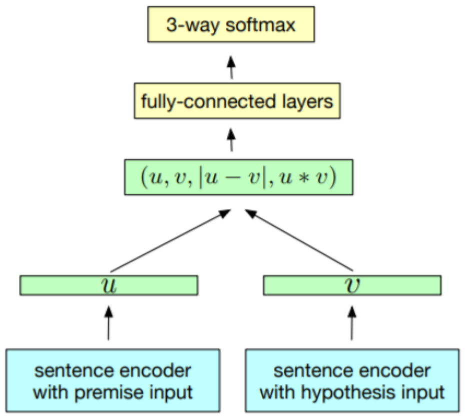
   - entailment classifier:
     after applying stage 1 on the premise and hypothesis sentence, we obtain 2 vectors, let’s denote them by u_p and u_h.
     They created a new vector m, $s.t.\ m = [u_p, u_h, |u_p - u_h|, u_p \otimes u_h]$.
     Next, m is fed into an MLP layer with a softmax layer for final classification

We wrote an implementation for this; the script can be found at stacked_BiLSTM.py
But unfortunately, we couldn’t continue it because of lack of ‘computer power’. Our laptop was way too slow for this one. We think that 3 BiLSTM’s are slowing down the process. In addition, they use Glove pretrained vectors, which are updated during training as well.
The next paper we introduce, ‘fixes’ these issues, it uses only one BiLSTM and the pretrained vectors are not being updated.

3. Supervised Learning of Universal Sentence Representations from Natural Language Inference Data by Conneau et al. '17 with an accuracy of 84.5%.
The reason for that one is, again, because of its simplicity. It's quite the same as in number 2, just simpler. After creating a script for number 2 and failing, it was easiest to pass to this one.
As before, the model is built out of 2 stages.
   - sentence encoder:
     They used a fixed word-embedding with pretrained 300D Glove 840B vectors. The word-embedded sentence they sent into a BiLSTM layer and performed row-max-pooling on the outputs of the BiLSTM, as described in the following image:



   - entailment classifier:
     They applied the sentence encoder on the premise and hypothesis sentences and obtain 2 vectors, u and v respectively.
     They created a new vector m, $s.t.\ m = [u, v, |u - v|, u \otimes v]$.
     Vector m was fed into an MLP with one hidden layer and a softmax for classification.


3-way softmax
fully-connected layers
$(u, v, |u - v|, u * v)$
u
v
sentence encoder with premise input
sentence encoder with hypothesis input

This was our final paper and it's script can be found in BiLSTM_max_pooling.py

The structure of this model was quite simple to implement.
BUT as we proceeded, we encountered some tremendous problems.

Difficulties + solutions

While implementing the above (equally for paper 2 and 3) we ran into some difficulties.

- Memory. The paper we worked on (all the papers we checked) uses pretrained 300D Glove 840B vectors. This file contains more than 2 million(!) vectors and it weighs over 5 GB. Our program crashed even before training started because of memory issues.
  We managed to solve that by creating a file containing vectors for the words found in our datasets. We have a script for that, my_glove_version.py
- Debug. In order to debug our implementation, we couldn't afford running our code for several hours just for it to crash for some exceptions (mostly dimension issues).
  So, we wrote a script to shorten the datasets, we used 0,001% (‼) of the datasets.
  The script can be found in reduce_sizes.py
- Running-time. The code would run for ever. At first, every example (out of half a million in the training set) would take a few 1-3 seconds to train.
  We tried several methods some of them really helped:
  1) First, we got to understand more how the computation graphs (cg's) work. In our first implementation attempt, we called npvalue() on a dynet expression which was totally unnecessary and way too early. This caused the program to forward calculate the cg for every example. (We did it because for beginners like us, it's easier to make mathematical calculation with actual values rather than dynet expressions.)
     So, we took the advice found on the site of DyNet and called forward on the cg as late as possible, namely at the end of each batch when updating occurs.
     By doing so, we saw great improvements in running time.

  2) We tried gaining some time by padding all the sentences in a batch so they should all be from the same length and hopefully the dynet's autobatch will speed things up. Although we knew we might lose accuracy, but it was worth the try. It took us a while just to get this to work. At the end, we did not notice much of a difference, so we did not continue with this approach.

  3) After running our model for several hours, we again encountered memory issues, while evaluating the model on the dev set. We think that having all the embeddings, the training set and the dev set in memory causes our program to crash.
     So we had an idea, which also gained us some running time.
     Instead of loading the whole train set to memory and then looping over it every epoch, we decided not to load it but rather read line by line during training itself.
     This saved us at least 20 minutes of loading time.
     The only downside of it, there's no way we can shuffle the training set before every epoch, which might lead to a slight loss of accuracy.

- <u>Saving completed work</u>. After experiencing several crashes, we had the idea to save the model at multiple point during training. We save the model at the end of every epoch.

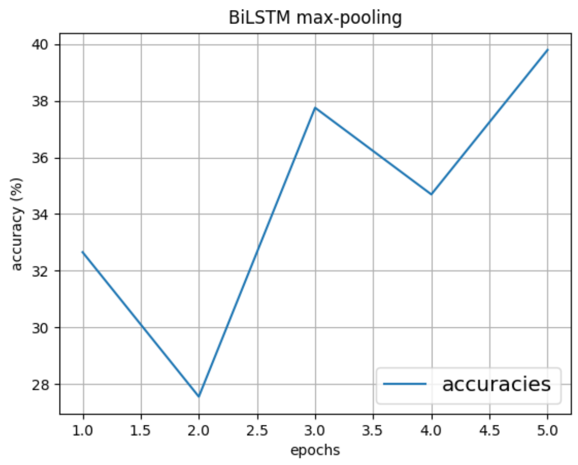
Training details

- we used SGDTrainer
- 10 epochs
- we started with a learning rate of 0.001 and halved it every second epoch
- embedding dimension is 300 (from Glove)
- the output dimension of the LSTM's is 512
- hidden layer in MLP of 512
- we used ReLu as activation function
- embeddings are not being updated

Results

Unfortunately, after all we tried, we still couldn't properly run our model ☹☹☹
We spent so many hours on that but couldn't get any of it.
After 5 hours of running, our model completed only 25% of a single epoch!
(We couldn't just let our computers run for 100 hours as we have multiple courses with programming exercises)

We did try to get something out of our model.
So, we created 3 datasets (train, dev and test) which contain 1% of the original data.
We successfully ran our model on these files for 10 epochs with SGDTrainer, but accuracy was bad (37% on dev set and 33% on the test set). At least we know why they need a training set with half a million examples.
We then tried again, but this time 5 epochs with AdamTrainer (learning rate 0.005) and results were quite the same as before, we got 40% on the dev set and 33% on the test set.
Below is the graph:



We regret we couldn't run our model on the full data, but for that we need more time and more powerful computers as well.