UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Machine Learning

Course Project:

Contextual LSTM models for Large scale NLP tasks

Authors:
Pol Alvarez Vecino
Olga Fetisova

Professor:
Lluís Belanche

June 27th 2017, Barcelona

Contents

1 Introduction

The Long-Short Term Memory model and its different variants have achieved impressive performance in different sequence learning problems in speech, image, music and text analysis, where it is useful to capture long-range dependencies in sequences. This observation motivated us to explore the use of topics of text segments to capture hierarchical and long-range context of text in LMs. A few papers describe a very detailed work performed on this current topic.

Specifically we will try to implement part of the experiments described in the paper "Contextual LSTM (CLSTM) models for Large scale NLP tasks" [1]. Contextual LSTM (CLSTM) are an extension of the recurrent neural network LSTM model, where we incorporate contextual features (e.g., topics) into the model. Further on, we evaluate CLSTM on an NLP task: **word prediction**. And finally, as the last step, we compare results with the original work.

The initial results from experiments described in the suggested research paper by Ghosh, run on English Wikipedia corpus of documents. They indicate that using both words and topics as features improves performance of the CLSTM models over baseline LSTM models for these tasks. For our project, we will compare our results with the given ones and draw our conclusions accordingly.

1.1 Goals

The basic objective of this project is to train a Contextual Long-Short Term Memory (CLSTM) neural network. However, the tool used to create this context/topic is not available due to intellectual property. We will tryto reproduce the paper using other available topic detection techniques and compare their performance with the original results on the word prediction task.

The project was developed incrementally with the following objectives as guide:

1. Pre-process dataset
2. Word embeddings training
3. Basic LSTM implementation
4. Context creation using different techniques
5. Contextual LSTM model training
6. Perplexity comparison of the results for the different models.

We started by obtaining the basic information required to train a simple LSTM. We started by getting the Wikipedia dump, preprocess it, and train word embeddings of the vocabulary to use them as inputs for the basic LSTM model (see Section 3).

The next step was to explore how to provide context/topic for a text fragment. During this phase we explored many topic detection methods (such as LDA and LSI) as well as other approaches (such as using as context an average of the word embeddings). We trained these topic models (the ones that required training) with the whole Wikipedia dataset and encapsulated them in order to use them during the network training.

1.2 Data

The dataset used for this project is the Wikipedia English snapshot of 20th of February 2017. It is a full raw dump so it contains HTML tags, links and many other useless informations that need to be stripped.

- Compressed data size: 13 GB

- Uncompressed data size: 57 GB
- wikiExtracted data size4: 12 GB
- Number of documents: 5.339.722

The proposed research project dataset varies from the our one because of the updated version of the Wikipedia data. For the Ghosh's experiments, they used the whole English corpus from Wikipedia (snapshot from 2014/09/17). There were 4.7 million documents in the Wikipedia dataset, which they randomly divided into 3 parts: 80% was used as train, 10% as validation and 10% as test set.

2 Related work

The most relevant related work for this project is the research paper *Context dependent recurrent neural network language model* [5] in which the authors based their work. Basically, the work by Ghosh et al. is an extension for LSTM models of the model proposed by Mikolov et al.: a recurrent neural network with context (CRNN). We find that both papers provide already related work enough so will not extend this section further. For more information, please check both related work sections of their research papers.

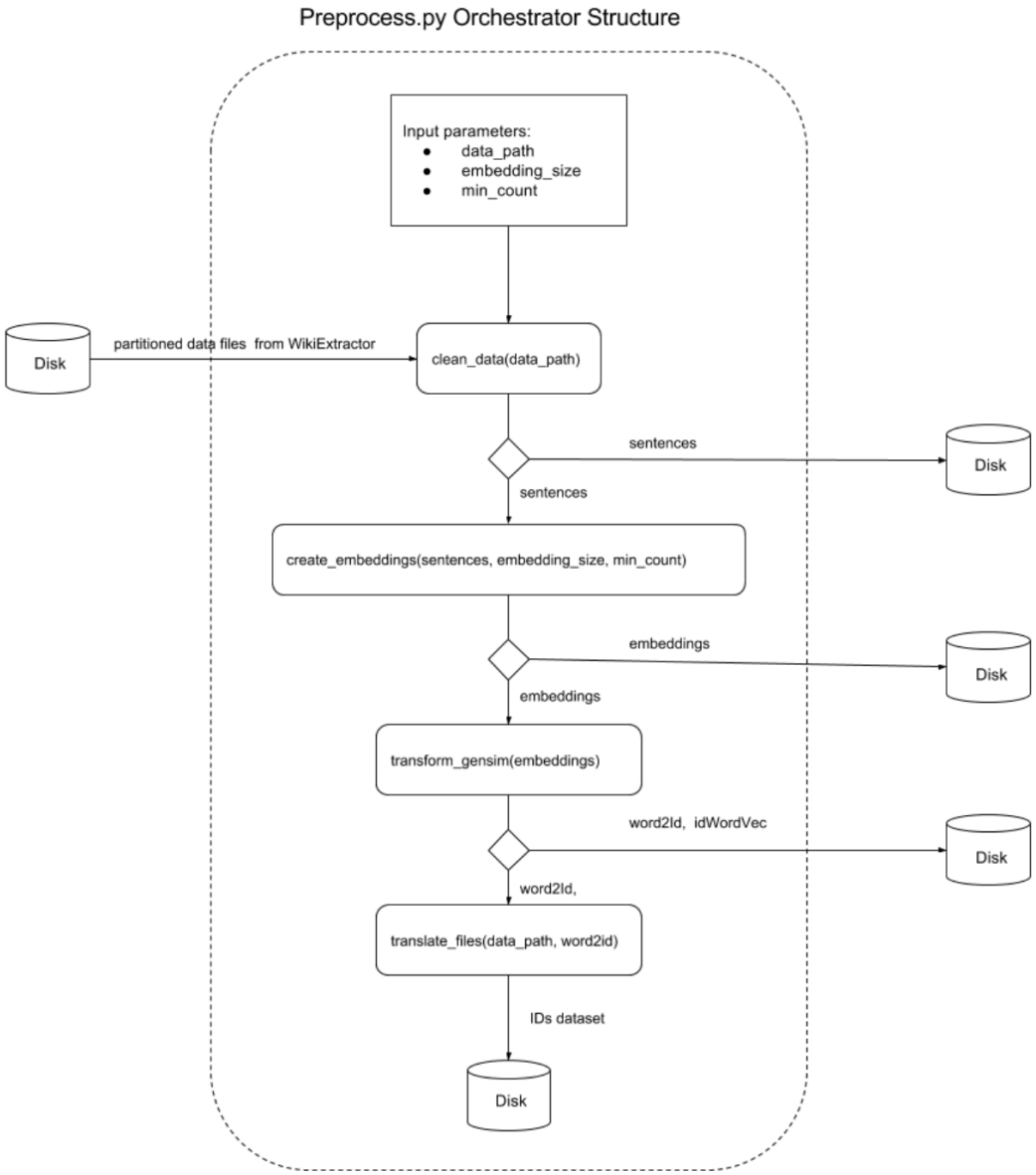
3 Data preprocessing

This section gives a brief overview of the preprocessing done to the Wikipedia dataset. It is essentially a summary of the document *Word Embeddings and Topic Detection for Contextual LSTM NLP tasks: Part I* under the documentation folder but contains some changes introduced by later stages of the project (such as more appropriated data structures).

Processing such a large dataset requires a lot of time. If all tasks are performed by a single application, any error or unexpected problem cause the loss in computation hours. In order to minimize these risks while enabling fast prototyping and development iterations, the whole task at hand was divided into smaller blocks to ease the process.

Blocks are structured in such a way that can be called as a function inside a pipeline defined by an orchestration file or as a standalone program. For each block, its results are stored to disk (even when they are in a pipeline) in order to be able to resume executions from the last failed block or to modularly modify and test each functionality.

Figure 1 shows the execution flow and generated data defined by the orchestrator file preprocess.py.



3.0.1 Extraction

- WikiExtractor, python script that extracts and cleans Wikipedia dumps

Prior to processing the wikipedia dump must be extracted. The data dump contains many XML tags and other useless (for our purposes) data that we would like to strip. Also, it is desirable to split the dataset into smaller files in order to parallelize the computation of next steps.

WikiExtractor project [1] is designed to do exactly that. It takes a wikipedia dump as input, cleans it of XML tags and links, and partitions it into 100 Mb files containing many articles. It is worth noting that it does not process the actual text in any way. Thanks to WikiExtractor the resulting files' size is 12 GB (which is less than the compressed dump file) instead of the 57 GB of the uncompressed data.

3.0.2 Preprocessing

- embeddings.py

[1] `https://github.com/attardi/wikiextractor`

It handles the preprocessing of the "wikiExtracted" data and creates the embeddings with it. Once the data is clean and in text format, the next step is to parse it into words. To choose the format we took into account two requirements: train the word embeddings with the resulting data and being able to feed the data into a neural network implemented in TensorFlow.

For the network training, we need to keep the article-paragraph-sentence structure. After evaluating many alternatives, the chosen format was raw text with the tags ¡eos¿ and ¡eop¿ delimiting end of sentences and end of paragraphs respectively.

The processing time of this section in sequential mode was roughly 22 hours. Using python's multiprocessing module [2], it was parallelized to use four times the number of available CPUs of the running computer thus allowing to switch context when threads are blocked in I/O (which is the most part of the time due to the read/write performed in each thread). The parallelization version took only 45 minutes using 16 CPUs. As a side note, multiprocessing's Pool was not able to manage memory efficiently so it crashed after some files were processed (with 120 GB available RAM memory). Simple Processes were used and their memory unallocated for each file. With the data divided into lists, the next step is to train the word embeddings.

3.0.3 Embeddings

- embeddings.py

handles the preprocessing of the "wikiExtracted" data and creates the embeddings with it Word embeddings are a group of NLP techniques that map words to numerical vectors. They can be used for language modeling, feature learning, and dimensionality reduction. There are many methods to construct this mappings but we used the Word2Vec implementation of gensim python package [3]. Word2Vec algorithm [4] trains a neural network and uses the weight of each word as embedding. Depending on the model, the task is different.

This project used Skip-gram architecture, shown in 2, in which the goal is to predict the surrounding words of the current word. Gensim implementation is based on [4] with the improvements introduced by [3], which obtains better performance and more regular representations with sub-sampling techniques and Negative Sampling.

The method used takes a list of documents. In our case each document was a sentence because word embeddings are done at sentence level (i.e. the sliding window starts and ends in each sentence). In order to train it, three parameters need to be set:

- **Embedding size**, number of weights per word during the training (i.e. final size of the word embeddings).
- **Min. word occurrences**, number of minimum occurrences per word to be included in the Skip-gram training.
- **Window size**, maximum distance between the current word and the word to be predicted.

The bigger the embeddings are the better because they have more expressive power. However, too large sizes will lead to long training times. The actual sizes is often decided empirically. For this project, three embeddings were computed with size 200, 500, and 1000.

The minimum occurrences per word was set to 200 because it is the value used in the original paper [1]. The window value was set to 10 (again this parameter is tuned empirically and general opinions regard 10 as an expressive enough value for posterior deep learning).

The embeddings training time was 3.5 hours when using vectors of length 200 and all the available CPUs.

[2] `https://docs.python.org/2/library/multiprocessing.html`
[3] `https://radimrehurek.com/gensim/models/word2vec.html`

4 Context

4.1 Topic analysis

This section describes the research done in order to find a plausible context creation technique for the CLSTM network. This explanation is a summary of the document *"Word Embeddings and Topic Detection for Contextual LSTM NLP tasks: Part II"* under the documentation folder. Refer to that document for a more detailed explanation.

Associated files:

- **src/context/topics_analysis.py,** handles the creation of the required inputs for each of the topic models.
- **src/context/creator.py,** utility class that groups all information required to produce a topic given a document and a method.

From the different context combinations shown in the paper we chose the best composed of:

SentSegTopic:
sentence segment topic, which is the topic of the current sentence segment read so far.

ParaSegTopic:
paragraph segment topic, which is the topic of the current paragraph segment read so far.

To produce this two kind of topics we need a method able to get a topic out of an arbitrary-long word sequence. Together with that, the next capabilities are also desirable:

- Evaluation of context for a sequence of words with arbitrary length
- Reasonable memory footprint
- Fast context computation
- Production of sensible contexts

In order to provide context for the posterior CLSTM implementation the first approach was to use topic detection (TD). TD methods are trained upon a corpus of documents and can then produce a topic for new documents (of arbitrary length). They fulfill all our requirements out-of-the-box quite well (except the ability to check if the topics are sensible for some of the methods). In exchange for this capabilities though, they have a long training time so we will need to see if they can produce useful topic models in a reasonable time for such a large dataset.

We explored the most frequently used algorithms for topic detection and evaluated their performance on large datasets. Specifically, the topic detection algorithms used were:

- Latent Dirichlet Allocation online [2]
- Latent Dirichlet Allocation offline
- Latent Semantic Indexing
- Hierarchical Dirichlet Process

Online LDA yielded good results in short time (12 hours). However, the parallel version of offline LDA finished also in a reasonable time (14 hours) and we observed more consistent results (probably because Wikipedia data has some topic drift, and LDA online assumes the corpus does not have topic drift). The results for the LDA implementation are quite easy to validate as they produce a sequence of words per topic with decreasing probability.

The LSI was quite faster (about 4 hours) but the results it produces are far more difficult to validate. They produce sums of a fixed number of words per topic in which the weights are positive and negative (not probabilities).

Hierarchical Dirichlet Process was the most promising method because it is non-parametric (for LDA and LSI we needed to specify the number of topics in a quite trial/error or arbitrary way) bayesian method. However, the implementation used is still under development and we did not manage to obtain sensible results.

4.2 Context creation

· **creator.py,** implements the context creation using LDA best word, LDA average and LSI sum methods (described below)

From the network point of view, passing the context just means a bigger input. Specifically, the input is three times bigger because we now pass along the embedding of the current word, the one representing the sentence segment context, and the one of the paragraph segment.

We proposed four different techniques of building such a context. The first two ones rely on LDA model, the third one uses LSI, and the last one is based on embedding space arithmetics.

First LDA approach was to simply use the word embedding corresponding to the most important word of the topic. We decided to try this simple approach because the small trials done with the LDA model reported sensible words for sentence and paragraphs segments. Second method used was to do weighted sum (using their probabilities as weights) of the first ten relevant words for the topic of the segment. This second method was added to check if some relevant information that one-word per topic technique could not provide is important for the word prediction task.

For LSA we decided to use the sum of the all words belonging to the topic because the number of words per topic is fixed. The weights are not probabilities so we are not just finding the "center" of the topics averaged by their relevance, instead, this is more similar to performing semantic arithmetics on the embedding space as we did in previous work (where for example King+Woman-Man=Queen). Because of that, it is more difficult to interpret the topics and harder to assess if the results are correct with respect to LDA.

Finally, we will also consider another method which does not require any further computation: averaging the embeddings read so far. For this technique we will start with the embedding of the first word as context. Next, for each new word we will average the current context with the next word embedding. We will try two approaches: one in which every time we compute the whole average so that all words have the same weight; on the other, we will just do the mean of the context and the new word, this will have the effect of overweighting recent words and making the older ones vanish. Both methods will be reset for each new sentence (when computing the SentSegTopic) or each paragraph (when computing the ParaSegTopic). Probably, averaging all the words will be better because we are working with sentences and paragraphs (which should be short) and having the focus always on the most recent words does not seem useful.

For more detailed analysis of the produced contexts check Section 7: Results of *"Word Embeddings and Topic Detection for Contextual LSTM NLP tasks: Part II"*.

5 LSTM & CLSTM

· **lstm_frag.py,** implements regular LSTM for word prediction

· **clstm.py,** implements Contextual LSTM for word prediction

The LSTM and CLSTM version designed are both based on the example TensorFlow tutorial which uses Penn Tree Bank Dataset [4].

Both networks use backpropagation through time (BPTT), gradient descent and dropout (to prevent overfitting and allow to use same network for training and testing [6]). The output variable is the word ID because TensorFlow can use the output without one-hot encoding (i.e. it handles the softmax layer and the loss computation together in order to do it more efficiently).

The model was modified to receive our trained embeddings (of size 200, 500, or 1000) as inputs for the LSTM; and to receive the embedding + sentence segment context + paragraph segment context for the CLSTM.

Additionally, CLSTM has an input parameter to choose which context method should be used:

1. LDA using best word as topic [lda]
2. LDA averaging best 10 words of topics [lda_mean]
3. LSI using sum of all topic words [lsi]
4. Arithmetic which sums all the words of the segment [arithmetic]

The networks have four configurations: large, medium, small, and test in order to facilitate the local testing and try different configurations. See *SmallConfig, MediumConfig, LargeConfig, and TestConfig* classes inside *lstm_frag.py or clstm.py* for more details.

6 Validation Protocol

We set up the same validation/testing protocol the paper describes. We split the whole Wikipedia into three datasets with 80% for training, 10% validation, and 10% testing. In the original research, validation data was used to tune the number of LSTM cells and topics added to the input data. In our case, we used the best parameters already, so the validation set's purpose would be to choose between the model used for topic creation (see Section 4.1).

The test set results are used to compare them to the paper results.

The metric used was word perplexity which is a fairly common choice for language models.

7 Results

This section presents the results of the execution of a regular LSTM with its training, validation, and validation perplexities and its learning rate.

The CLSTM was far too slow to even finish a single epoch. We present the evolution of the CLSTM's training perplexity against a regular LSTM as a proof that the network is working and that its evolution shows promising results.

Figure 1 shows the train, test, and validation perplexities of a LSTM network. Figure 2 shows the corresponding evolution of the learning rate used for the gradient descent during the training. The correlation is clear, the smaller the learning rate the slower the improvements.

[4] https://www.tensorflow.org/tutorials/recurrent

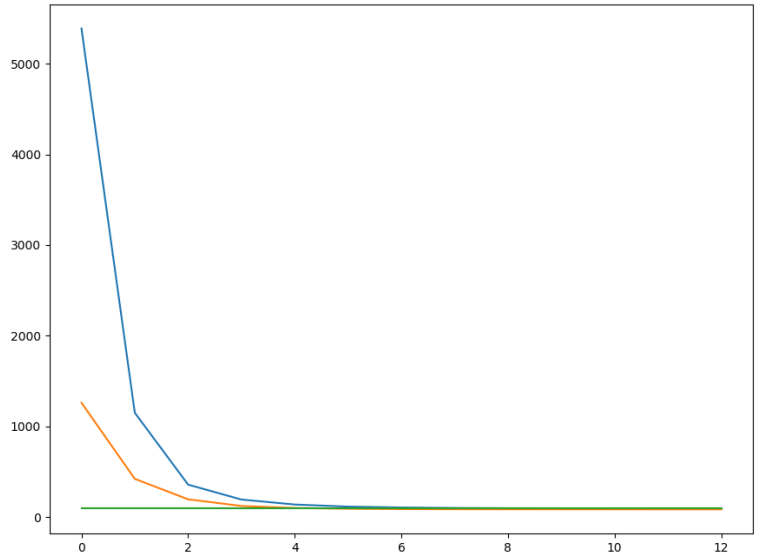


Figure 1: Training (blue), testing (orange), and validation (green) perplexities of a LSTM network through 12 epochs. Validation is only computed once the training is finished. Validation has not been actually used for model selection so it is possible to compare the training perplexity against it.

.

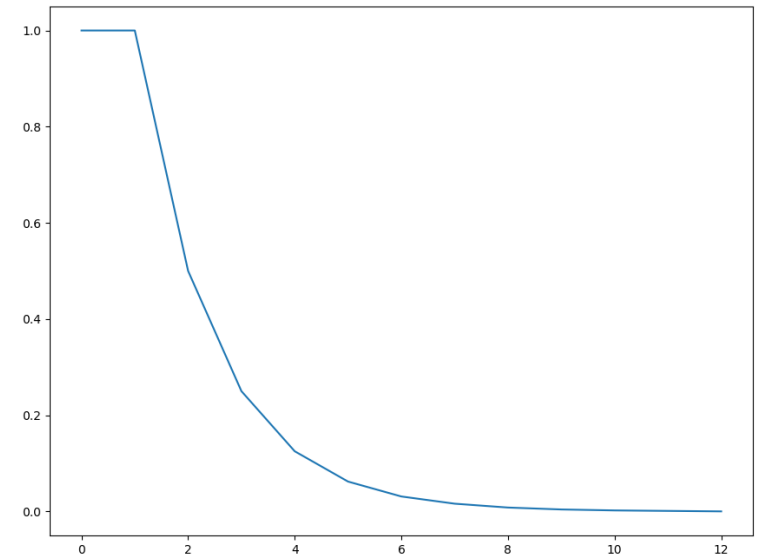


Figure 2: Learning rate evolution during the execution of a LSTM network with Medium configuration and 12 epochs.

.

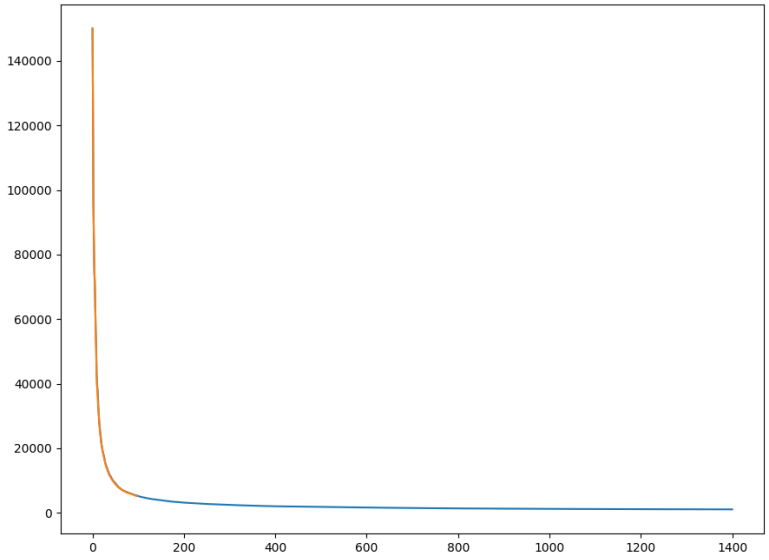


Figure 3: Training perplexities of a CLSTM (orange) and a regular LSTM (blue). They seem to be superimposed until the moment were we do not have more CLSTM data. Figure 4 shows a zoomed in version of this plot.

.

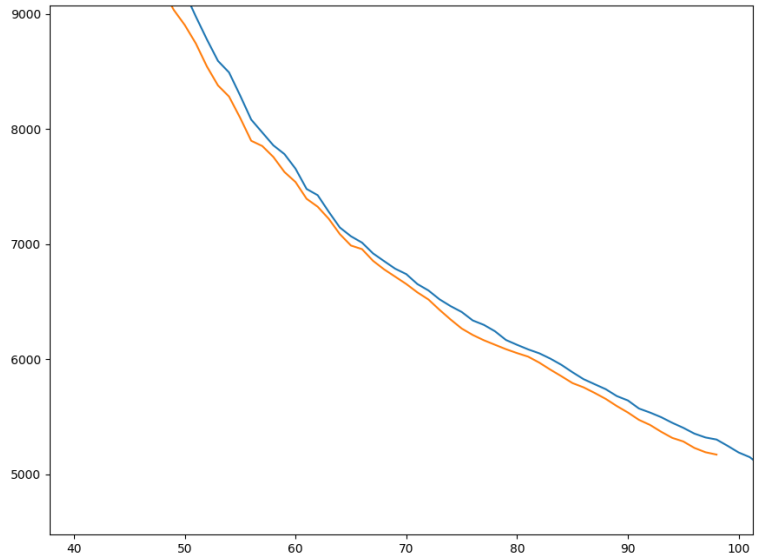


Figure 4: Detail of Figure 3. Again, in orange CLSTM and LSTM in blue. We can appreciate here that CLSTM training perplexities are actually lower than LSTM. Although not conclusive without validation/testing results it certainly shows promise and makes it interesting to keep working to improve CLSTM training times.

.

8 Conclusions

Due to the time constraints we have not managed to throughly test the implemented CLSTM models. We have barely managed to provide to insight about how the LDA-based context creation using a single word affects the performance of the LSTM model. We have greatly underestimated the amount of time necessary to train a model, and specially the complexity of optimizing it for TensorFlow (TF), and the restrictions imposed both at hardware and software level.

First LSTM model was implemented quite fast but TF does not allow tensors bigger than 2Gb (and our dataset in integer format was ¿5Gb). We had to rework all the input pipeline in order to split the dataset into smaller chunks and feed them to the model. However, this has slowed down the model training quite significantly.

With respect to the context creation, the use of different frameworks (TF and gensim) and methods (embeddings vs. topics) caused many integration issues and extra overhead (i.e. needing to translate between the mappings of the embeddings and the topics).

9 Future Work

Future work is basically divided into two great topics. First, we will work to reduce the training overhead because, even with great resources, deep learning is out of reach (specially with datasets this big) if all the training pipeline is not heavily optimized. Second, once the model achieves feasible training times we will actually study the different topic models and try to tune their parameters such as: augmenting/reducing the embeddings size, the number of topics for LDA/LSI, the number of words considered per topic, have different weights when averaging the seen embeddings as context, and many more details which should be decided empirically.

9.1 Reducing training time

In order to greatly reduce the I/O overhead the first method is to load the data from disk asynchronously. TF already provides some sort of API to do it, but extending our implementation with a custom thread that loads all the data would not be too difficult.

Next, we will study how to better parallelize the actual graph computation. TF provides some intra-node parallelism (by using MKL) but is not able to exploit full resources out-of-the-box (as for example the 4 available GPUs per node instead of just 1).

Once optimized the training for a single-node, the next step would be to distribute the training. This is usually done by training multiple instances of the same model at once and applying the average of all gradients. Another method, is to have a single "model" server which receives gradient updates concurrently.

Further improvements will be based on using TensorFlow to automatically train the word embeddings on the fly, using many under-the-hood optimizations. In this case, we will need to check that all words provided as topic are part the TF input dictionary.

9.2 Context

Once the model is fast enough, we would like to optimize the context provided. We will try to threshold the number of words/topics used when averaging LDA depending on their probabilities (with some metaheuristic).

It will also be interesting to check how many codification decisions actually affect the network. For example, we have decide to provide the embedding corresponding to the unknown words when an adequate enough topic was not found, but we could also provide a zero valued, vector or even provide the previous one.

With respect to the actual models used, we want to explore why we did not get any good results out of the Hierarchical Dirichlet Process. We believe it was by far the most promising because not having to specify the number of topics allows to reuse it give any given dataset instead of requiring manual tuning. Also the original paper used "Hierarchical Topic Model" as fake name to refer to their actual topic.

Finally, if the speed up is not enough to compute the models online we will compute them offline and annotate the dataset with them, trading computation time for I/O time (which we already described how to optimize until it is "negligible").

References

[1] Shalini Ghosh, Oriol Vinyals, Brian Strope, Scott Roy, Tom Dean, and Larry Heck. Contextual LSTM (CLSTM) models for Large scale NLP tasks.

[2] Matthew D Hoffman, David M. Blei, and Francis Bach. Online Learning for Latent Dirichlet Allocation. *Advances in Neural Information Processing Systems*, 23:1–9, 2010.

[3] Tomas Mikolov, Greg Corrado, Kai Chen, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, pages 1–12, jan 2013.

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. oct 2013.

[5] Tomas Mikolov and Geoffrey Zweig. Context Dependent Recurrent Neural Network Language Model. 2012.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.