Georgian Language Model

Aleksandre Khokhiashvili Tamta Topuria Archil Ksovreli Bakari Gamezardashvili

{alkhok18, ttopu18, aksov19, bgame19}@freeuni.edu.ge

Abstract

We implemented a Georgian language model, which generates Georgian text according to the given sentences.

Our final model is a BERT transformer trained from scratch with custom configurations. Using MLM (Masked Language Modelling) as well as NSP (Next Sentence Prediction) while training. We tried training this model with word2vec pretrained subwords, however it didn't give us better results.

We built an n-gram base model to compare the generated results. As evaluation metrics we used a perplexity indicator, which we also modified for subwords. As well as manual evaluation.

We trained our best model on the nVidia Tesla K80 GPU for 10 hours. We think the results can still be improved with further training.

1. Introduction

NLP tasks in the Georgian language are a very interesting problem. 2021 graduates of our university have tried making a good language model for the Georgian language, many of them succeeded. However there is still minimal open-source research in this area.

Quality Georgian language model would be a real asset to many fields. Chatting bots, fraud detection, writing..

With this paper we want to focus on exploring paths of next word generation as well as masked language modeling. To achieve this goal we chose transformer architecture. Transformer architecture has shown really good results on language modeling tasks in various languages. So we couldn't resist trying it on Georgian.

We measured the results by generating text with Beam search coupled with Top-k. Manually looking at the generated text. Also by computing the perplexity on the test set and comparing it to the benchmark model's results.

We learned that transformer architecture really learns meaning in the language. However it needs much more training to give satisfactory results.

2. Data Collection

We used already collected data from the CC100[1] database. This was constructed using the urls and paragraph indices provided by the CC-Net repository by processing January-December 2018 Commoncrawl snapshots. Since the CC100 data was mainly from the Georgian online news outlets, it was fairly political, yellow press and lacked academic structure. We wanted to have a diverse dataset, including formal written text, with good grammatical structure. For this reason, we enriched our dataset with the books written in foreign languages and then translated to Georgian[2].

By combining the datasets our model was trained both on modern Georgian day-to-day language and formal literature.
[Data collection done by Archil Ksovreli and Tamta Topuria]

Figure 1: Punctuation marks are common as expected, however not as common as they were before filtering the data (dots were extremely common before). Opening parenthesis is as common as closing parenthesis, which is valid. Notice the big comma, which will ruin our first transformer model. Next biggest words are linking words, which is also natural. Then comes Georgia, its capital city, journalistic narration words, etc.

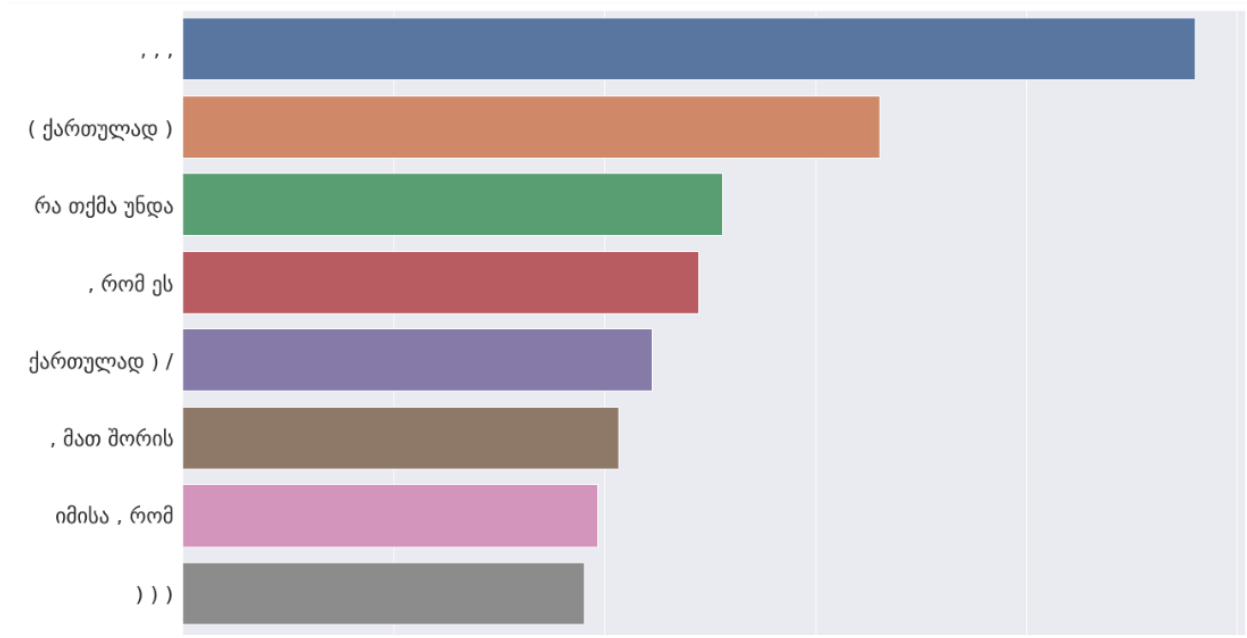


Figure 2: Trigram analysis of our data shows that three dots is the most common phrase, which is ok, but the ratio of how common it is seems a little concerning. Then comes “(ქართულად)”, which we think is a result of scraping Georgian movie sites, they have little description of the movies and then language of the dubbing. After that is “რა თქმა უნდა”, which is indeed a very common phrase and so on.

3. Data Analysis and Filtering

First of all, we wanted to validly recognize sentences from the data. We couldn’t do it just by splitting the text on end-of-sentence punctuation marks (. ? !). Because some Georgian words include these marks. So the first thing we did was identifying and altering these words: ძვ.წ. → ძვ<prd>წ<prd>, dates (20.12.2013 → 20<prd>12<prd>2013)...

In our final model we also removed all non-Georgian characters to get better results.

To analyze the data we modeled our text as a WordCloud presented in Figure 1. It gave us a good understanding of the most common words in our data.

We also modeled trigrams shown in Figure 2 to see which phrases were most common. Results were somewhat satisfactory.

[Filtering done by Aleksandre Khokhiashvili, Archil Ksovreli, Bakari Gamezardashvili]
[Modeling done by Tamta Topuria]

4. Models

4.1 Word Embeddings

We wanted to try training our models with and without pre-trained Word2Vec embeddings. As there are no such embeddings for Georgian corpus, we trained Word2Vec ourselves with the gensim library's gensim.models.Word2Vec function.

At first we generated embeddings for words, but then for better performance we opted for subwords in language models. So we had to generate subword embeddings to use in those models.

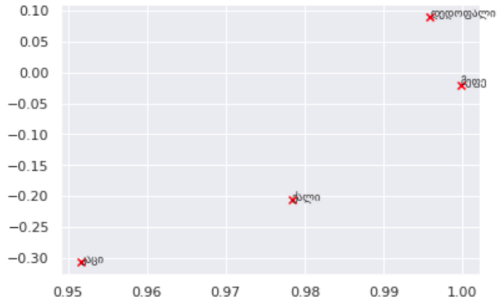
These are results of word2vec trained on words.

```
model.wv.most_similar("ბრწყინვაღე")
```

```
[('შესანიშნავი', 0.45644110441207886),
 ('დიდებული', 0.43956077098846436),
 ('ჩინებული', 0.438639760017395),
 ('დააგვირგვინა', 0.4363951086997986),
 ('მაზიარა', 0.4328196942806244),
 ('ღვთიური', 0.4310285151004791),
 ('საუცხოო', 0.4308733344078064),
 ('მშვენიერი', 0.43077394366264343),
 ('დახვეწილობით', 0.42676424980163574),
 ('რეჟისურით', 0.42587947845458984)]
```

```
model.wv.most_similar(
    positive=['ბავშვი', 'დიდი'],
    negative=['პატარა'])
```

```
[('ადამიანი', 0.35643741488456726),
 ('ხანია', 0.33954620361328125),
 ('ბავშვის', 0.33606427907943726),
 ('უდიდესი', 0.32642653584480286),
 ('აუნაზღაურებელი', 0.3238275349140167),
 ('რამხელა', 0.31815198063850403),
 ('არასრულწლოვანია', 0.31586235761642456),
 ('სადისტური', 0.31162646412849426),
 ('ბავშვები', 0.30134543776512146),
 ('ხარჯვაა', 0.29946595430374146)]
```



Additionally, we have a genius idea for a new tech product: "Predict your perfect partner using word2vec embeddings!". See sample results below:

```
model.wv.most_similar(
    positive=['მშვიდმსანი', 'ცოდი'],
    negative=['ქმარი']
)
```

```
[('კირჩხიბი', 0.575498104095459),
 ('ღრიანკაღი', 0.537632405757904),
 ('კურო', 0.5362414717674255),
 ('მორიელი', 0.5302557349205017),
 ('ვერძი', 0.5221663117408752),
 ('მერწყული', 0.5191493630409241),
 ('შეთავსების', 0.504426896572113),
 ('ურთიერთნდობა', 0.5035413503646851),
 ('სასწორი', 0.48609426617622375),
 ('თევზები', 0.4859570562839508)]
```

```
model.wv.most_similar(
    positive=['ბაქარი', 'ცოდი'],
    negative=['ქმარი']
)
```

```
[('ზურაშვიდი', 0.5313971638679504),
 ('კაკიაშვიდი', 0.5305842161178589),
 ('ნატუკა', 0.5259012579917908),
 ('ზურა333', 0.5243187546730042),
 ('ღევანიკო', 0.5229456424713135),
 ('ჟივიძე', 0.5224558711051941),
 ('საშვედები', 0.5186235904693604),
 ('ზაქაიძე', 0.518191397190094),
 ('ბუღრიძე', 0.5173429250717163),
 ('მრევლიშვიდი', 0.5168269276618958)]
```

We think word2vec learned the essence of the Georgian language and got pretty good analogies.

[Word2Vec training and modeling done by Tamta Topuria]
[Adjusted to subwords by Archil Ksovreli]

4.2 Subwords

We chose to try training our models on subwords because they make a lot of sense for a Georgian language model, due to georgian grammar and structure. There are many forms (ბრუნება) of the same words: ვჭამე, შეჭამე, შემომეჭამა. Dividing these in subwords means that our model might learn meaning of these suffixes and prefixes and of course cores (ფუძეები) of words. So if there is ვჭამე but not შემომეჭამა in the text, model will still learn meaning of შემომეჭამა, as it probably learns embeddings for “შემო”, “##მე” “##ჭამ” “##ა”
(“##” means that the subword is in the middle or at the end of the word)

```
wb_tokenizer.encode('შემომეჭამა').tokens
```

```
['შემომ', '##ეჭ', '##ამა']
```

We set a maximum number of subwords for the algorithm to generate so it unites the subwords, which are often used together. That’s why “შემო” and “მ” are united here.

```
wb_tokenizer.encode('თმათეთრი').tokens
```

```
['თმა', '##თე', '##თ', '##რი']
```

```
wb_tokenizer.encode('შავთეთრი').tokens
```

```
['შავ', '##თე', '##თ', '##რი']
```

```
model.wv.most_similar('##ები')
```

```
[('##ებიც', 0.6364842653274536),
 ('##ების', 0.49715572595596313),
 ('##ებისთვის', 0.48254188895225525),
 ('##ებისგან', 0.46793222427368164),
 ('##ურები', 0.42871180176734924),
 ('##ერები', 0.41608086228370667),
 ('##ებია', 0.4138261675834656),
 ('##კები', 0.41119346022605896),
 ('##ეები', 0.4096052944660187),
 ('ღირ', 0.4076472520828247)]
```

```
model.wv.most_similar('ღორ')
```

```
[('ძროხ', 0.8630732893943787),
 ('##ატარე', 0.8326977491378784),
 ('##ყრიდი', 0.8293331861495972),
 ('ძაფ', 0.8266956210136414),
 ('მღვ', 0.8252092599868774),
 ('ქუდ', 0.8243218660354614),
 ('ღიღ', 0.8230547308921814),
 ('გადაქ', 0.8226965069770813),
 ('##აღაგ', 0.8214482665061951),
 ('ღოყ', 0.819601833820343)]
```

Word2Vec on subwords.

```
wb_tokenizer.encode('მაგარია ძააააააან. ძვ.წ.').tokens
```

```
['მაგარია', 'ძა', '##აააა', '##აა', '##ნ', '.', 'ძვ', '.', 'წ', '.']
```

[Subwords done by Tamta Topuria and Aleksandre Khokhiashvili]

4.1 Benchmark Model

As a benchmark model we trained an N-gram.

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. A good N-gram model can predict the next word in the sentence i.e the value of p(w|h)

So N-gram only looks back at n-1 words. Then computes the probability of the n'th word according to the previous n-1 words. These probabilities are generated from the frequencies of N-grams in the train data.

We used nltk library's nltk.lm.model[3] to get our N-gram model.

For n-gram on words our parameter n was 3, but for subwords we increased n to 6. Reason for this is simple: subwords tend to be shorter than words.

```
rds=20, text_seed=['ეს', 'არის'], random_seed
```

```
'ეს არის ფინალისტების ყველაზე მოკლე სია ბოლო წლების განმავლობაში.'
```

We got pretty good results on the few and frequent phrases in terms of readability and meaning. Because it just found this phrase in train data and continued as it was written there.

```
text_seed=['ჩემი', 'ხატია', 'სამშობლო'], rand
```

```
'ჩემი ხატია სამშობლოების, იმპერიალისტური სახელმწიფოების, და უწინარეს ყოვლისა დასავლეთისგან.'
```

On longer and less frequent phrases which were not in the train set N-gram showed its weaknesses in terms of generating meaningful sentences. We could see that N-gram didn't grasp the language.

```
when using the same parameters.
seed=['ქართველი', 'ერი', 'ამას', 'არ', 'აიტანს'],
```

```
'ქართველი ერი ამას არ აიტანს მას სართულებზე.'
```

Here N-gram looked at "არ აიტანს" and generated "მას სართულებზე" accordingly. If it had realized earlier context of "ქართველი ერი ამას", It would've guessed that "აიტანს" had different meaning here. But it only looks at n=6 previous subwords. These examples show the shortcomings of N-grams.

[Benchmark model done by Archil Ksovreli]

4.2 GPT style transformer

Transformers have caused a real revolution in modern NLP. It is a deep learning model that adopts the mechanism of self-attention, differentially weighting the significance of each part of the input data.

Because of its full understanding of the given context (due to the attention mechanism) we thought the transformer model would learn better than recurrent models. It would be faster due to parallel computing capabilities and lack of long dependencies would improve performance.

We wanted to have full control on the layers and implementation so the first transformer model we constructed ourselves according to the PyTorch manual. As a training task we used next token prediction, each input is a token and the same index on output is trained to guess the next token.

We trained this model for 2-3 hours. Which is not much but loss was not decreasing anymore and manual evaluation results were also bad. It **always** generated the same tokens no matter the given sentence.

```
batch_id = 29 | cur_loss = 8.637796401977539 | elapsed tim
e = 4.442430019378662
eval(ჩემი აზრით) = ჩემი აზრით , , , , , , , , , ,
eval(დღეს გარეთ ძაღიან წვიმს. არ დაგავიწყდეს) = დღეს გარეთ
ძაღიან წვიმს. არ დაგავიწყდეს , , , , , , , , , ,
```

Here are beginnings of the two example sentences: “ჩემი აზრით” (in my opinion) and “დღეს გარეთ ძალიან წვიმს, არ დაგავიწყდეს” (It’s rainy today, don’t forget the). Model predicted commas on both of them.
We believed that the main reason for this was that our model could achieve relatively good loss by outputting the most common token (comma). Due to this, our model was getting stuck in this terrible local minimum.

We should mention that training with pre-trained Word2Vec embeddings gave no better results.

[First transformer done by Aleksandre Khokhiashvili, Archil Ksovreli and Bakari Gamezardashvili]

4.3 BERT transformer

Following our previous result we thought about introducing a different task which would not be as easy to cheat for our model. By cheat we mean getting a good loss by just outputting the most common token.

BERT is trained using two tasks: masked language modeling(MLM) and next sentence prediction(NSE). NSE tasks output is a single probability which can’t be solved by trivial methods like before. Because of this we thought our model would need to get at least some understanding of language to make progress on NSE tasks. Once it would start understanding language then MLM taks would solve the more important task for us.

Since BERT is a fairly complex model, first we tried to find a PyTorch implementation that we could use manually. But unfortunately there were none. Because of this we opted for using huggingface, which is not ideal since we lose a lot of control of what actually goes on inside the model. Even when using hugginface we found out that most of the guides online pre-train only one of the two tasks. But luckily we were able to finally find a way (TextDatasetForNextSentencePrediction) and tried training our models. huggingface library also allowed us to train our model using subwords pretty easily, which is amazing for Georgian language, as explained above.

We soon realized that the model was not great after short periods of training. Looking at the BERT paper we noticed that with our GPU it would take at least 2 weeks to train fully. So we let our model train for 10 hours and actually got some plausible results. This time model was learning embeddings for itself.

With not much time left before the presentation we switched our gears and tried to train our model using Word2Vec embeddings for subwords as well. We ripped our word embedding layer from BERT and replaced it with a pretrained one. It's important that we did not freeze our weight because otherwise the model would not be able to modify [MASK], [SEP], [CLS] token embeddings which would be devastating. After about 6-7 hour training we noticed that results were not even close to the previous model so we gave up on it. It's worth to note that during this second phase we also filtered out foreign letters from data.

After this we also tried 2-3 hour training with this new data but we didn't get anything of value.

[BERT done by Aleksandre Khokhiashvili and Tamta Topuria]

5. Evaluation

For the evaluation section we'll focus only on our best model. This was a BERT model trained on NSE and MLM tasks with the first version of data (foreign languages not filtered out). Trained for 10 hours.

5.1 Perplexity

One of the best demonstrations that our model was actually understanding language was perplexity values on different valid and invalid sentences:

```
get_perplexity('ჩემი აზრით ეს არის ძალიან კარგი იდეა')
```

```
224.1802215576172
```

```
get_perplexity('ჩემი ძალიან ეს არის აზრით იდეა კარგი')
```

```
3592.35595703125
```

```
get_perplexity('ჩემი ჩემი ჩემი ჩემი ჩემი ჩემი ჩემი')
```

```
7492.962890625
```

```
get_perplexity('ჩემი და რომ მაშინ რადგან რისთვის იმიტომ')
```

```
1698.384521484375
```

We can see that perplexity is significantly lowered by properly ordering the given sentence. Additionally, just using high frequency words does result in good perplexity, unlike models with small training time. This demonstrates that the model understands context, order and meaning.

Model also does quite well in some cases when tasked to fill in one masked word in a sentence.

```
fill('პრეზიდენტი მიხეილ [MASK] აზრით ამ შეხობის ამეხება კარგი იდეაა.')
```

```
[{'score': 0.047650258988142014,
  'token': 5596,
  'token_str': 'საკაშვილის',
  'sequence': 'პრეზიდენტი მიხეილ საკაშვილის აზრით ამ შეხობის ამეხება კარგი იდეაა.'},
```

Text generation is not ideal probably because BERT is not directly trained to solve this task, but we believe that this could be much better with a few weeks of training. (or TPU training like the BERT paper).

```
beam_search('დღეს მე ვერტმფრენით გავფრინდი', num_tokens = 10)
```

```
['დღეს მე ვერტმფრენით გავფრინდი და აი, შენ არ ვიცი, რა არის ეს',
 'დღეს მე ვერტმფრენით გავფრინდი და აი, შენ არ ვიცი, რომ იმიტომ რომ',
 'დღეს მე ვერტმფრენით გავფრინდი და აი, შენ არ ვიცი, რომ იმიტომ,',
 'დღეს მე ვერტმფრენით გავფრინდი და აი, შენ არ ვიცი რა უნდა რომ ასე',
 'დღეს მე ვერტმფრენით გავფრინდი და აი, შენ არ ვიცი, რომ ასე ვარ']
```

```
beam_search('პრეზიდენტი მიხეილ', num_tokens = 10)
```

```
['პრეზიდენტი მიხეილ ზურაბიშვილი : გიორგი ვაშაძე, პრემიერ - მინისტრის საგარეო საქმეთა',
 'პრეზიდენტი მიხეილ ზურაბიშვილი : გიორგი ვაშაძე, პრემიერ - მინისტრმა საგარეო საქმეთა',
 'პრეზიდენტი მიხეილ ზურაბიშვილი : გიორგი ვაშაძე, პრემიერ - მინისტრის შინაგან საქმეთა',
 'პრეზიდენტი მიხეილ ზურაბიშვილი : გიორგი ვაშაძე, პრემიერ - მინისტრი საგარეო საქმეთა',
 'პრეზიდენტი მიხეილ ზურაბიშვილი : გიორგი ვაშაძე, პრემიერ - მინისტრის საქმეთა საგარეო']
```

Perplexity for n-gram on test data was 1062 and our model got perplexity of 2886. This is not ideal but fixable with training time. This assumption is believable because our model got significantly better each hour.

[Perplexity on all models done by Bakari Gamezardashvili and Aleksandre Khokhiashvili]

5.2 Language generation: Beam Search and Top-k

Beam search is the most popular search strategy for the sequence to sequence Deep NLP algorithms.

The beam search algorithm selects multiple alternatives for an input sequence at each timestep based on conditional probability. The number of multiple alternatives depends on a parameter called Beam Width B. At each time step, the beam search selects B number of best alternatives with the highest probability as the most likely possible choices for the time step.

Greedy Search will always consider only one best alternative. So beam search is always better than a greedy search. However in our case beam search was prone to generating repetitive sequences. It is also fairly slow if the beam width is big because of the exponential time complexity.

Hence we decided to also use the Top-k method. Top k sampling means sorting by probability and zero-ing out the probabilities for anything below the k'th token. It appears to improve quality by removing the tail and making it less likely to go off topic.

We also used repetition penalty so our model would be less likely to generate meaningless sentences like: “ჩემი ჩემი ჩემი ჩემი…”

[Beam search done by Aleksandre Khokhiashvili and Bakari Gamezardashvili]

[Top-k done by Archil Ksovreli]

5.3 BERT and N-gram comparison

The N-gram model has random results on OOV (out of vocabulary words). Because N-gram works with the probabilities of the words. If it has never encountered given words, the model has no data about the word’s probability. BERT model doesn’t really care if the word is OOV or not. This model works on understanding the language, not by statistically predicting. However there are no OOV words in our models, because we trained them on subwords. So if the word is a complete mess each letter will be a different subword. We can

still construct OOVs by using Latin characters though.

Grammatically speaking neither N-gram nor BERT are looking good. BERT really makes use of punctuation, but it still needs more training to master the Grammar. With N-gram there is no further hope.

6. Conclusion

Our number one lesson was that training transformers, or any mode, takes days even with a good GPU. We learned how important the training objective of the model is. How to guide the model with additional tasks to truly learn the language and not by achieving a small loss with cheating around local minima. In this case we had to give our model two tasks: next sentence prediction and masked language modeling. This way it was more concentrated on the context and true meaning of the text. We also learned how the basic model (N-gram) gave us good, at first glance even better results than the complex transformer. But the longer and more complex the input sentences became, the more obvious it was that N-gram was just choosing the best words according to the last few subwords and had no idea about the meaning. While the transformer struggled we see it as a positive that with more and more training it got significantly better and it really showed the understanding of the language.

Generally, during the project, our focus was to try and actually make a good Georgian language model. We tried to understand and think about the consequences of each line we wrote for each of our models. That is why our evaluation might be a bit lacking, because most of our energy went into the training. But we hope our attempts and different strategies of training were not in vain.

References:

1. CC100 Dataset: https://paperswithcode.com/dataset/cc100
2. Books written in foreign languages and translated to Georgian: https://drive.google.com/drive/folders/1guEMNElOnfjJvjv0UiqYuZ5cyC7xudeN
3. N-gram model from NLTK:

https://www.nltk.org/api/nltk.lm.models.html
4. Attention is all you need: https://arxiv.org/pdf/1706.03762.pdf
5. PyTorch LM from scratch example: https://github.com/pytorch/examples/tree/main/word_language_model
6. BERT paper: https://arxiv.org/pdf/1810.04805.pdf
7. BERT model by huggingface: https://huggingface.co/docs/transformers/model_doc/bert

[Report done by Tamta Topuria and Aleksandre Khokhiashvili]