Presentation slides:
https://docs.google.com/presentation/d/1nc3rH3LBKAREtRoAhkXyzJzGAxbUTrApVsTcmq-5lpU/edit#slide=id.g5b1a94b322_0_27

Modeling the Source Code

ECS289G Final Project Report

Tianqi Zhu, Scott Madera

1. Task description including motivation

In this project, we are going to model the source code based on a recent approach [1]. We use Byte Pair Encoding (BPE) data and train it with Recurrent Neural Network (RNN). After the training stage, we apply the beam search to search for the best K predictions of the next token. With a small train on a JAVA BPE data with around 2000 word vocabulary, we achieved a similar mean reciprocal rank (MRR) score with the original paper. We also investigated the influence of beam search size K. Finally, we apply the model to deal with pure English word tokens and find the MRR score decrease a lot. For more chart and data, refer to the presentation link shown at the header of the first page.

The motivation for this project comes from three parts. First, we want to model the source code as predicting source code automatically and suggesting source code for programmers by machine learning are exciting tasks. With the development of machine learning and computer architecture, we can now achieve higher accuracy on such tasks. Second, we want to verify the approach and generate MRR to see how good the model can be. Third, we want to practice our skills to implement algorithms from paper.

2. Prior work in the area, and the precise contribution of your project, and (if applicable) how it goes beyond

The prior work we mainly referring to is the paper we mentioned above. In that paper, they use BPE to solve the out of vocabulary (OOV) problem, which is that new identifier names are continuously invented by developers [2]. Traditional language models limit the vocabulary to a fixed set of common words and generate bad predictive performance. Thus, they introduce a new open-vocabulary neural language model, using BPE to represent subwords as segmentation into tokens. To generate BPE subwords, a BPE expansion algorithm is applied [3]. The key fact of BPE data generation is the number of expansions. According to their result, larger BPE subwords data with more expansions performs better. They adopt gated recurrent units (GRUs) for their language model and finally, they use the beam search algorithm to predict best K tokens.

In our project, we extend our given basic model to this paper's model. The contribution of our project are:

1. We implement the beam search on the basic model and investigate the beam search size's effect on prediction performance
2. We generate and implement an algorithm to measure MRR of beam search result
3. We reproduce equal level accuracy comparing with the original paper on a JAVA small train
4. We test model's MRR dealing with only English word subwords

For the first contribution, we implement the beam search, using two priority queues method. The effect of size K on beam search in terms of MRR can be concluded in Fig 1. This figure is measured on a model trained with around 30M tokens JAVA BPE data with around 2000 subword vocabulary. We test the MRR using 100 sequences and generate 5000 measure positions. We can see that the beam search (K = 3,5) has a positive effect on MRR comparing with no beam search (K = 1). However, larger K does not guarantee to have better performance in terms of MRR as we can see large K (K = 10) perform worse than small K. We considering this analogy to how much longer or shorter memory we want to use. When K is large, more long term memory may affect the prediction and cause a decrease in the rank of the target word.

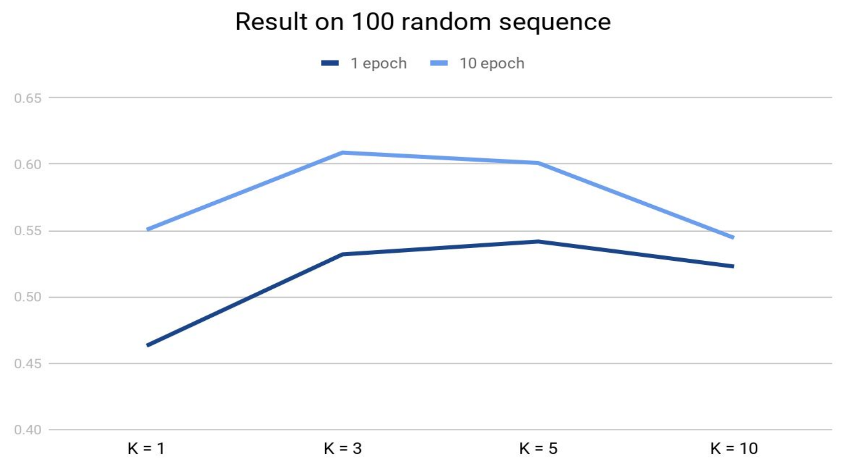

Fig 1. The effect of beam search size on MRR

For the second contribution, we come up with an algorithm to measure the MRR of beam search output. As the batcher function from the basic model can generate the sequence of subwords, we find all end-token subwords in a sequence and record the subsequence from beginning to the end-token subwords. The gap between two adjacent subsequences is the word to predict for the fist subsequence as the gap is exactly a token. We then can feed the first subsequence to the beam search and then measure the MRR score using beam search result and the target token generated by the gap.

For the third contribution, the paper achieves 62.87% MRR using a 2000 expansion BPE JAVA data with 15.74M training tokens. We achieve 60.87% MRR using a 2066 vocabulary BPE JAVA data with around 30M training tokens.

For the fourth contribution, we train the model on pure English subwords and test MRR after. We get a MRR of around 20%, which is significantly lower than the MRR for complete BPE data. This suggests that the model's correct prediction on structural subwords like {}, (), makes up a big part of MRR.

3. What Data you gathered, and how you did that. Refer to the diagram below

For this project, we don't collect any data except the BPE data provided in the basic model. The reason comes from three parts. First, the provided data is exactly what we need. Second, even the provided BPE small train data takes a long time to train and test; if we collect a big train data, the time to train and test will be out of control. Third, our aim of the project is more focused on paper algorithm implementation. Although we could have our own BPE algorithm, we believe this algorithm is well implemented already. However, for the beam search, there is no workable beam search on the basic model and thus we focused on the beam search.

Nevertheless, we still include a typical flow chart of the data gathering process for this model as a reference, which is shown in Fig. 2. The flow chart starts from JAVA Raw. We need to first apply the BPE algorithm to generate JAVA BPE data (as those given in basic model). After that, we will index map the BPE so that we can get batch data, which will later feed into the neural network or other tasks.

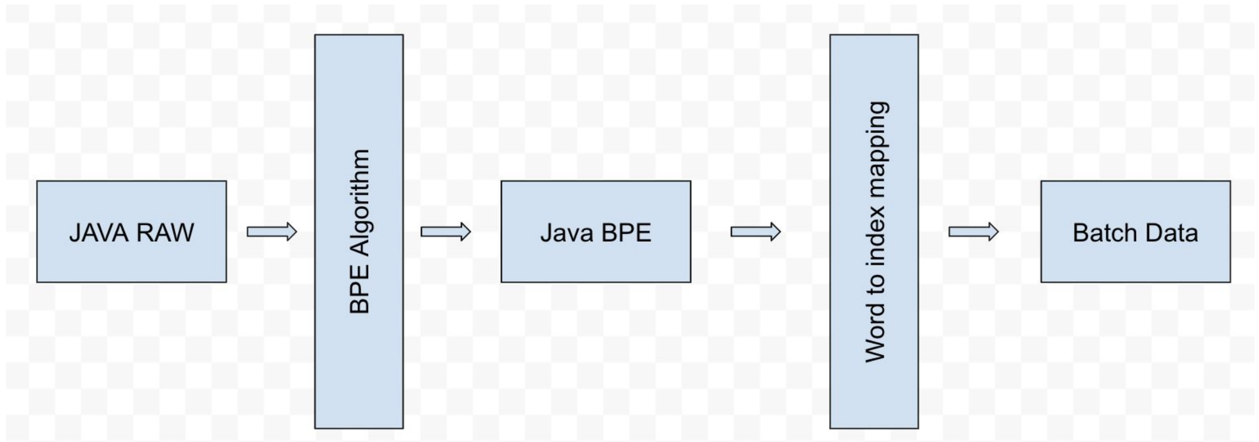

Fig 2. Data Processing Graph

4. Carefully document your work so far, so someone has a chance of picking it up

The model.py code in the GitHub repository is commented. The detail to run the code is recorded in README.md. We use Google Cloud to run the model.

The data folder contains training and validation and test BPE data. Training BPE data is larger than 100MB and is stored with git lfs.

The code folder contains the code for the model. The structure is similar to the basic model. We mainly code in model.py, and a little bit on data_reader.py. There are some other files to store the model and data states. A detail list and functionality of important files is shown below:

1. model.py: the file we implement all algorithms. To train the model, one needs to comment out the training part of the main function and change the test flag. The model and batch data will be saved to a given file. To test the model's MRR, one needs to comment out the test part of the main function change test flag. The model and batch data saved will be loaded from the given path. When test, one can modify the test sequence size, beam search size, and random seed. Because we test the model on a saved model and saved batch data, with the same random seed, we can generate the same test data and thus dame MRR score.
2. data_reader.py: the file we modified to train and test pure English subwords. To use, simply comment out the original batcher function and rename the batcher_only_alpha to batcher. Then train and test normally.
3. temp.txt: the output or terminal tee into this file
4. store_1.pckl: the batch data of an 10 epoch training, currently serving as our main test data. This file is handled by git lfs as it's around 400MB.
5. test_model_*: the saved model of a 10 epoch training, currently serving as our model.

There are mainly two algorithms in model.py. One is the beam search, and another is measuring MRR. The detail implementation of two algorithms can be found in the presentation link at the header of the first page.

The structure of the neural network is shown in Fig. 3. It is base on the current config.yml file. The batch data is embedded into 1000 dimensions and then feed into two-layer GRU units. The output GRU units then go through a drop out layer and finally maps back to the original vocabulary size by a dense layer. This is a typical RNN language model except the batch data is BPE data. Finally, the output of the neural network will be passed into the beam search and measure for MRR.

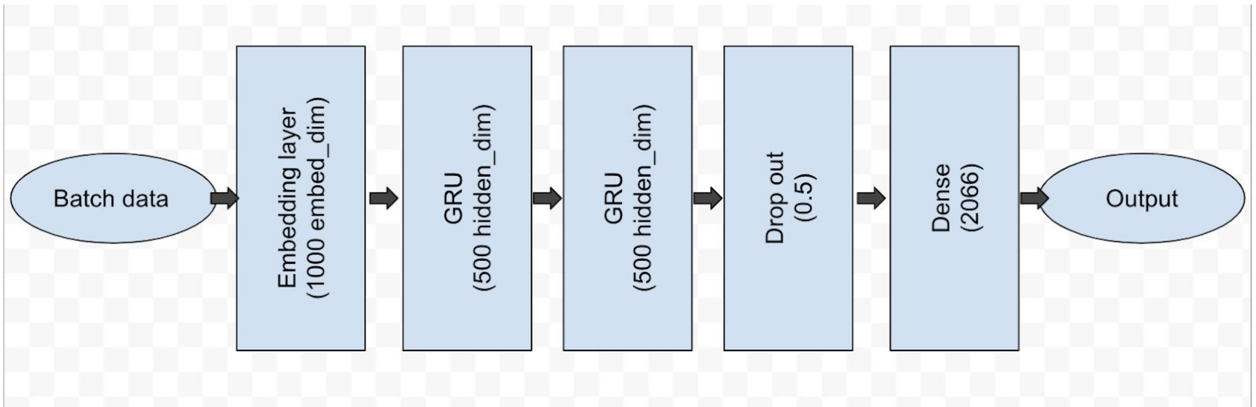

Fig 3. Machine Learner

We don’t really record the shape of data flow or tensors. On the one hand, to get shape is pretty straightforward by using the print function. On the other hand, at the beginning of this project, we took understanding the shape of the data flow task as a practice to understand the model, which we found super useful. Thus we consider leaving this task as a practice for people who really want to pick up the project.

5. Bibliography

[1] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. CoRR abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[2] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR ’13). IEEE Press, Piscataway, NJ, USA, 207–216. http://dl.acm.org/citation.cfm?id=2487085.2487127

[3] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. CoRR abs/1508.07909 (2015). arXiv:1508.07909 http://arxiv.org/abs/1508.07909