Joint Embeddings of Programming and Natural Language

Joseph Pagadora
Stanford University
jcp737@stanford.edu
SUNet ID: jcp737

1 Problem Description and Challenges

Natural language processing has come a long way and has made significant progress in the past few years. This project attempts to explore and improve on the problem of searching through code semantically through natural language queries. This is very useful as some people might want to search through code written in a language that they are not familiar with. Furthermore, this may make searching for particular code functions or snippets easier by avoiding the need for exact keyword match. An interesting thing to note with this particular problem is that code is *not* natural language, so this poses a challenge for this work.

2 Dataset

In this project, we only used a subset of the Code Search Net Corpus from [1], [2]. Recall that the entire dataset contains approximately 2 million pairs, each consisting of a function (written in Python, Java, Go, Javascript, PHP, or Ruby), and its documentation string (docstring). The subset of the data we used consists of just the pairs written in Python, in addition to other (code, docstring) pairs. The total amount of data we used here for training is about 800,000 points. The data is scraped from popular GitHub repositories. A challenge in this problem, as mentioned in [2], is that a function's documentation might be biased in the sense that it is inherently different in purpose from the queries that seek the function in question.
As discussed in Section 3, we essentially use BERT to create joint embeddings of code and docstrings, and we use sequence classification as our surrogate task. The data is labelled 0/1, whether the given code function and docstring are related, i.e., the docstring describes the function. The test-set data is divided into batches, where each batch contains 1 million (code, docstring) pairs. Each batch is further divided into groups of 1000 such pairs, where the first in the group is a "relevant" pair, and the other 999 are "irrelevant" pairs.

3 Model

Recall that BERT is trained using a large corpus of data, but because the domain here is very specific, we must perform pre-training in order for our BERT embeddings to work well. Recall that the architectural idea of BERT is that it is in general a 12-layer neural network consisting of consisting of transformer blocks and attention heads. In our project, we instead use the specialized RoBERTa framework [8], the "robustly optimized BERT pretraining approach," which is an optimized version of BERT that is trained using much more data. In particular, we fine-tune it to the classification task outlined in the section above. Thus, the final layer of our modified RoBERTa is simply a linear dense layer followed by a sigmoid. Note that a consequence of this task is a *joint embedding* of both

CS230: Deep Learning, Fall 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

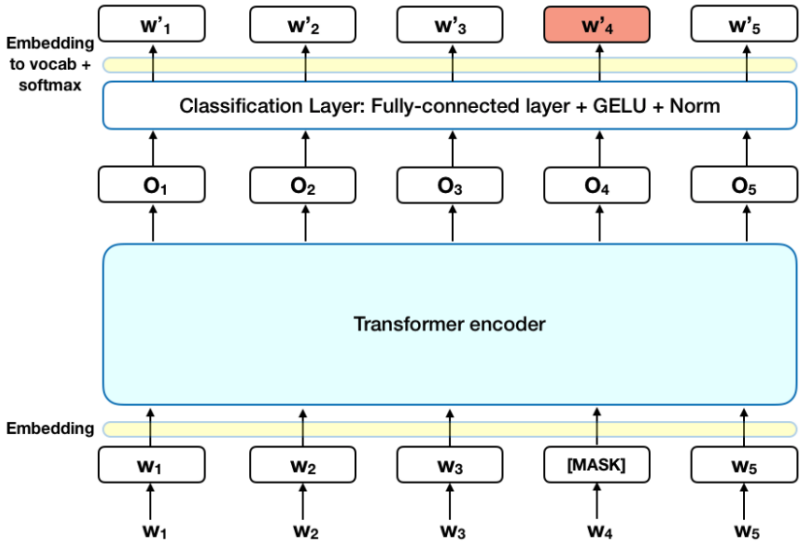

Figure 1: A sequence $\mathbf{w}$ is input into the model, with w_4 masked with the `[MASK]` token. The model then outputs a prediction for w_4.

programming language and natural language into the same vector space, which was just a hidden state within the RoBERTa model.

The data is preprocessed as follows. Recall that each datapoint is a (code, docstring) pair. As done with inputs to BERT, a `[CLS]` token is placed before the start of the string. Then, `[SEP]` tokens are placed in between the code and docstring portions of the data and at the end of the text. Then, we use the RoBERTa tokenizer to tokenize the text. See [6] for more details on how BERT tokenizes text data. Also similar to BERT, we create masks of either 0's and 1's, as these are also parts of the input to the model, intuitively, to "group" the tokens into the programming language group and the natural language group. Finally, the tokens, masks, and labels are all used as inputs to the RoBERTa model.

The RoBERTa model in this project was pre-trained, as described in [1] in more detail. Similar to how BERT was trained, the CodeBERT model in [1] was trained to optimize for two objectives: (i) masked language modeling, and (ii) replaced token detection.

3.1 RoBERTa Pre-Training

In objective (i), about 15% of the tokens are masked, i.e., replaced with a special `[MASK]` token. For each datapoint $\mathbf{x}_i$, suppose m_i tokens were masked out, leaving a modified sequence of tokens $\tilde{\mathbf{x}}_i$. The goal of the network p_1^θ with parameters θ is to predict the tokens that were masked out by minimizing the following loss function (which is just the negative log-likelihood):

$$\mathcal{L}_1(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i}\log p_1^\theta(\hat{y}_{ij}|\tilde{\mathbf{x}}_i),$$

where $\hat{y}_{ij}$ denotes the predicted token for the j-th masked-out token in the i-th example. The general model architecture for RoBERTa is shown in Figure 1, taken from [7]. Note that this is actually a diagram for BERT but the idea is essentially the same.

Objective (ii) is similar to objective (i) but with the difference being that masked tokens are instead replaced with different alternative tokens. In fact, this objective uses techniques similar to that of GANs. Two generators, one for natural language, and the other for programming language, are used to generate these alternative tokens, with a discriminator labelling whether each word is *different than the original* or not (as opposed to real or fake).
For each example $\mathbf{x}_i$, suppose m_i of its tokens were masked and changed to a different token by the generators, yielding a modified sequence $\tilde{\mathbf{x}}_i$. Let x_{ij} denote the j-th token in the sequence $\mathbf{x}_i$, and let $\delta_{ij} = 1$ if $\tilde{x}_{ij} \neq x_{ij}$, and 0 otherwise. That is, δ_{ij} is the indicator for whether the j-th masked token in the i-th example indeed changed values. Let p_2^θ denote the discriminator network, parametrized by θ, that predicts the probability that a given masked token was changed to something different than the

original. Its goal is to minimize the following loss function (just cross-entropy loss):

$$\mathcal{L}_2(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i}\Big[\delta_{ij}\log p_2^\theta(\tilde{x}_{ij}) + (1-\delta_{ij})(1-\log p_2^\theta(\tilde{x}_{ij}))\Big].$$

We want the entire RoBERTa model, parametrized by θ, to be good at both of these tasks simultaneously, so the total objective function will be

$$\mathcal{L}(\theta) = \mathcal{L}_1(\theta) + \mathcal{L}_2(\theta).$$

3.2 RoBERTa Fine-Tuning

Unfortunately, this pre-trained model does not perform well on the task of code search. Fortunately, however, all it takes is some fine-tuning on the labelled dataset, described in Section 2, to train a better-performing network in the task of code search. This particular model had exactly 124,647,170 trainable parameters. We performed fine-tuning on this model with mini-batch gradient descent with a batch-size of 100, an Adam optimizer, and 8 epochs. We used a learning rate of 1e-5 and an Adam ϵ of 1e-8. In addition, we used a maximum sequence length of 50. That is, after tokenizing the (code, docstring) string, we padded the sequence to length 50 with 0's if the length was less than 50. Otherwise, we truncated the tokenized sequence so that the final input sequence into the RoBERTa model had 50 tokens. We took into account the lengths of the tokenized code and tokenized docstring portions and removed tokens one-by-one depending on which portion was longer. This entire process to fine-tune RoBERTa, with the given architecture and hyperparameters, took approximately 8 hours on a small P3 AWS EC2 instance, which had only one NVIDIA Tesla GPU and 8 virtual CPUs.

4 Evaluation

Search engines indeed are scored using the *mean reciprocal rank* (MRR). If we have N queries, where potential answers are ranked in order of relevance, then the MRR is defined as

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}.$$

Thus, $0 < MRR \leq 1$. Using batch0 and batch1, we get MRRs of about 0.76 and 0.78, respectively. This competes very well with the MRR reported in [1], which for Python was 0.8685. Note that, according to [1], a CNN and a BiRNN give MRRs of about 0.57 and 0.32 for Python respectively. Furthermore, the performance of the base pretrained RoBERTa network (before any fine-tuning) gives MRRs of about 0.01 for batch0 and batch1.

Since each programming language is different, in order to extend this application to other programming languages, we would need a different fine-tuned RoBERTa model for each. Indeed, the performance of RoBERTa after fine-tuning seems to vary depending on the programming language. From [1], for instance, this model gives an average MRR of 0.6926 for the Ruby programming language. [1] also shows empirically that the joint tasks of masked language modelling and replaced token detection improves the performance for each programming language

5 Analysis and Future Work

Clearly, we can move on to fine-tune RoBERTa on the other five programming languages to get a better sense of its performance for general programming languages. Indeed, it may be possible to perform a n-way joint embedding of natural language and programming language, for any given $n-1$ programming languages. I had initially hoped to build an application for this task, but unfortunately due to my limited resources, this was not possible in the given timeframe. Thus, a clear future goal, practically-speaking, is to build such an app that indeed takes in natural-language queries and outputs code-snippets that are ranked by relevance. Furthermore, we can always continue to explore more various types of architectures, and even simply test out/reproduce this work using the simpler architectures such as nBOW, CNNs, or LSTMs, but it may be unlikely to perform better, both theoretically and practically, than the BERT models shown here.

As an analysis of results, the network we proposed here did not perform as well as the model proposed and fine-tuned in [1]. However, there is no doubt that this recent work has made great progress and achievement in this niche application. As mentioned above, the RoBERTa model fine-tuned here already performs much better than the "traditional" architectures, including bag-of-words (nBOW), CNN, BiRNN, and even Self-Attention. These model architectures gave average MRRs of about 0.58, 0.57, 0.32, and 0.69, respectively on Python test sets, according to Table 2 in [1]. Indeed, what is interesting is that the Bidirectional RNN seems to perform the worst with respect to MRR among all other models and architectures.

References

[1] Zhangyin Feng, Daya Guo, et. al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In *arXiv:2002.08155* (2020).

[2] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *arXiv:1909.09436* (2020).

[3] Sean Robertson. "NLP From Scratch: Translation with a Sequence to Sequence Network and Attention." From *https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html*

[4] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, Satish Chandra. "When Deep Learning Met Code Search". In *arXiv:1905.03813* (2019).

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. "Attention is All You Need". In *arXiv:1706.03762* (2017).

[6] Chris McCormick. "BERT Word Embeddings Tutorial". From *https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial*

[7] Rani Horev. "BERT Explained: State of the art language model for NLP". From *towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270*

[8] Yinhan Liu, Myle Ott, et. al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In *arXiv:1907.11692*