Towards Accessible Frameworks in Machine Learning

Cem Koc*, Eric Liu*, Jiayi Wang*, Yujie Xu*, Alvin Cheung

Department of Electrical Engineering and Computer Sciences

University of California, Berkeley

{cemkoc, eric.sa.liu, jiayi_wang, yux121, akcheung}@berkeley.edu

* authors contributed equally to this work

Executive Summary

Since 2012, the field of Machine Learning (ML) has enjoyed exponential growth in applications as well as theory partly due to the advances in computing and big data. As more and more people get interested in machine learning, the application space has been expanding from classical areas within ML (e.g. image classification, natural language processing (NLP), robotics, etc.) to biology (e.g. protein folding, disease detection), astrophysics (e.g. detecting new exoplanets) and many more. All of these are only possible because people in different fields take the time and effort to learn to write ML code using a plethora of complex frameworks and in many cases paired up with software engineers to get their code to work. At its core, our work aims to democratize this field by lowering the barrier to entry which will help attract more people from different backgrounds so they can have an easier time learning to write ML code using one of the most popular frameworks: PyTorch.

In order to achieve this goal, we finely-tuned neural code search models to allow users to search for PyTorch code semantically using natural language (NL) queries with which we hope developers can use to have an easier time when learning to write PyTorch code and applying ML to new problem domains. In order to develop a new language model (or fine-tune an existing one which is the approach here), we first needed to find a large dataset. Even though there are datasets available containing large amounts of Python source code, there is none whatsoever for PyTorch source code which is a *domain specific language* (DSL) for ML applications written in Python language. Therefore, we needed to learn novel

data engineering and data science methodologies to create a modern *extract-transform-load* (ETL) data mining pipeline with which we could collect a large corpus of NL and PyTorch (Code) descriptions. First, we identified four diverse set of sources from which we can mine the information we need to create this corpus. Afterwards, we proceeded to create automated scrapers and mining algorithms/heuristics curated to mine the information we were looking for while respecting users' privacy to the largest extent we were capable of doing which included: user id obfuscations, aggregations and dropping sensitive fields such as fine-grained timestamps from the final dataset. After extracting a raw dataset, we cleaned and curated to create a dataset with which we can fine-tune two different model architectures for neural code search task. We are hoping to extend this dataset and publish it to attract more ML researchers and engineers to find new ways to create better user experiences as well as tooling around writing machine learning code. Potentially, this dataset can be expanded and a new, larger model can be trained to generate working code examples from natural language queries. Moreover, a fine-tuned model can be frozen and deployed in a web server which can return PyTorch code snippets relevant to users' queries.

1 Introduction

1.1 Background

It has been shown that deep learning has potential to influence a wide variety of problem domains. From classical problems of image classification[1] and natural language processing (NLP)[2] to modern applications in genomics[3] and autonomous vehicles[4], deep learning continually blows away benchmarks and establishes itself as the new state-of-the-art approach. We expect deep learning to expand to solve new problems as technology improves and with it the number of machine learning (and specifically deep learning) engineers to increase.

One issue that is rarely discussed, however, is the complexity of learning and applying deep learning. Even beyond the highly theoretical and also mathematical foundations of deep learning, there are obstacles an ML engineer has to face when applying deep learning. With the large number of frameworks currently available (enough to warrant more than a couple "Top 10" lists) and complex and often dry documentation, actually writing deep learning code can pose a challenge. Engineers currently only have documentation, forums, and colleagues as sources of knowledge upon which they can build.

A search tool focused specifically on deep learning frameworks will provide immense value to programmers of all levels. It has been shown that code examples are an extremely valuable teaching technique[5]. Examples, even without context, provide more value as they show different ways to use similar concepts. We strive to show that this is possible for deep learning frameworks, a novel problem due to the

library/package nature of deep learning being a subset of a larger programming language.

PyTorch has truly become one of the most popular ML frameworks if not the most. A simple search on https://paperswithcode.com shows just how ubiquitous PyTorch has become[1] among the ML community. Therefore we choose PyTorch as our target framework, as not only it is one of the most widely used deep learning frameworks but also one that has had minimal API changes over its lifespan. As our problem takes natural language input and searches related PyTorch code as output, this can be seen as a sequence classification problem which is a well-known NLP challenge where the task becomes give a joined pair of sequence A (NL words) and sequence B (Code tokens) classify it as a match or not. Generally, solving this problem consists of two-steps, dataset collection (for transfer learning) and language modeling (fine-tuning).

As we will discuss more in the related work section, there was no existing NL to PyTorch dataset. We emphasized quality in our process, as creating a high quality dataset establishes a high quality foundation for any work built from the dataset. As one part of this effort, we chose four sources for our data: GitHub[2], PyTorch Discussion Forum[3], StackOverflow[4], and JuICe[6]. By having a wide variety of sources we hoped to capture as much existing PyTorch code as possible. We utilized custom built web scrapers and a unified ETL pipeline to assemble and clean our dataset. Our raw dataset from source included 68,398 natural language and PyTorch code pairs. Data cleaning that occurred after is discussed in the relevant modeling section of the paper.

To conduct our language modeling, we fine-tuned two models: CODEnn, an embedding based language model, and CodeBERT, a transformers based model. Our fine-tuning and training processes are described later in detail.

1.2 Related Works

As programming plays more important roles in the modern era, programming assistance technologies such as code search and code generation were investigated deeply by researchers. Code search model enables programmers to retrieve the pre-existing source code by inputting natural language descriptions, and code generation model helps programmers generate codes from scratch. As far back as 2015, researchers from the University of Washington developed the model CODE-NN[7] to perform the code search task on C# and SQL. In 2018, another CODEnn[8] model from the Hong Kong University of Science and Technology extends the code search task to Java. On the other hand, due to the introduction of Abstract Syntax

[1] Accessed May, 2021
[2] https://github.com
[3] https://discuss.pytorch.org
[4] https://stackoverflow.com

Tree (AST) in the paper from the University of California-Berkeley[9], code generation tasks extended to multiple programming languages. Specifically, RAT-SQL[10] generates SQL code, TranX[11] generates Python code, and CodeBERT[12], the current state-of-the-art, can generate Python, Java, JavaScript, etc. One recent work by Microsoft has begun establishing baselines for general code search, code-to-text, and other tasks. CodeXGlue [13] shares our vision for improved programmer productivity but doesn't have the deep learning scope which we are targeting.

Since most of the common languages like Java or SQL have been deeply investigated, there exist corresponding large-scale datasets. For example, the Spider dataset annotated by Yale students is a complex text-to-SQL dataset in various domains. WikiSQL is another semantic parsing dataset composed of questions and SQL code queries[14, 15]. In addition, the *CODEnn-Java-Train* dataset is a publicly-released corpus that contains Java code snippets and the corresponding descriptions[8]. However, when it comes to the PyTorch-related dataset, there are no existing resources that match our expectations. Accordingly, we collected the natural language-to-PyTorch corpus ourselves.

2 Dataset Collection

2.1 Diversified Data Derivation

Collecting high quality, curated datasets from scratch is a challenging and tedious work[16]. We looked into code search/generation papers and found that most researchers used multiple sources of datasets to ensure the generalization of their models or results. For instance, in one of the SQL generation papers [10], they used the Spider dataset for most of their experiments as well as conducting trials on the WikiSQL dataset to confirm generalization to other datasets. Another paper using deep neural networks to search Java/Android code included three training corpora for their experiments [17]. In addition to importing the CODEnn-Java-Train dataset mentioned earlier, they manually collected more Android-specific corpora from Github and StackOverflow[17]. Inspired by these papers, we decided to collect our PyTorch dataset from multiple sources so that our model would be robust and more generalizable.

We chose PyTorch Discussion Forum, StackOverflow, GitHub, JuICe as sources to obtain natural language and PyTorch code pairs. NL and code can be paired efficiently in question and answer (Q&A) platforms where NL is the question and code is the answer. PyTorch Forum is the official Q&A forum for ML/DL developers to discuss the use of PyTorch. StackOverflow is the biggest Q&A site for programming problems and includes a large amount of PyTorch-related problems. Besides the direct pairing in Q&A, NL and Code pairing can occur between any natural language description and code, therefore, general Python files and Jupyter notebook files are also valuable sources. GitHub is the biggest open-source

community for codebases, containing a tremendous amount of python files where any descriptive inline comment and corresponding code can be a good pair for our purpose. Finally, JuICe is a huge dataset of Jupyter notebooks and pairings can be made between inline comments and code or descriptions in markdown cells and code.

StackOverflow is a programmer website that shares insights and solutions to the posted questions in different engineering domains. Many researchers like the authors of "When Deep Learning Met Code Search" take advantage of Q&A resources on Stackoverflow to generate datasets for training [17].

PyTorch discussion forum is an open question asking/answering platform where users can both ask and answer questions on various topics and, depending on the quality of the answer, receive likes from the community. The discussion forum's structure shares lots of similarities with StackOverflow where a topic or a question can have multiple responses to it and a single solution is chosen among them to be "accepted".

GitHub is the biggest open-source community for code bases. The search engine of GitHub returns 61,278 repositories as the result of searching "PyTorch". Python files which include "import torch" are of our interest, and they can include inline comments and code following afterwards.

Machine learning engineers also tend to use Jupyter notebooks for exploration and analysis. Since Jupyter notebooks are comprised of cells, each of which can be marked as either natural language or code, notebooks are a common trend in creating reproducible data science [18] and unique for our purposes. JuICe is a 2019 dataset created from sampling a random subset of all "valid" notebooks from Github in addition to a hand curated selection of notebooks.

2.2 Data Processing for Dataset Curation

PyTorch Forum PyTorch discussion forum unfortunately does not provide any public APIs or datasets that would make it easier to query the question data on the website so we developed an automated crawler that would crawl the website and scrape question contents. Utilizing these scrapers, we first extracted the question URLs and split all the questions into two buckets: solved and unsolved with solved referring to those questions that have an official solution tag. This made it easier for us to develop appropriate heuristics for extracting natural language - code pairs for each bucket. We extracted title-code pairs from the posts within a question and filtered those that had less than 10 characters in their title. For the unsolved questions bucket, we picked the code which belonged to the post with the most likes. One observation we had when scraping PyTorch discussion forum is that people tend to post a lot of error outputs they received during runtime and since these are not useful for us we filtered and discarded them by simple keyword match. At the end of this process, we reached a total of $8,006$ title-code pairs for the

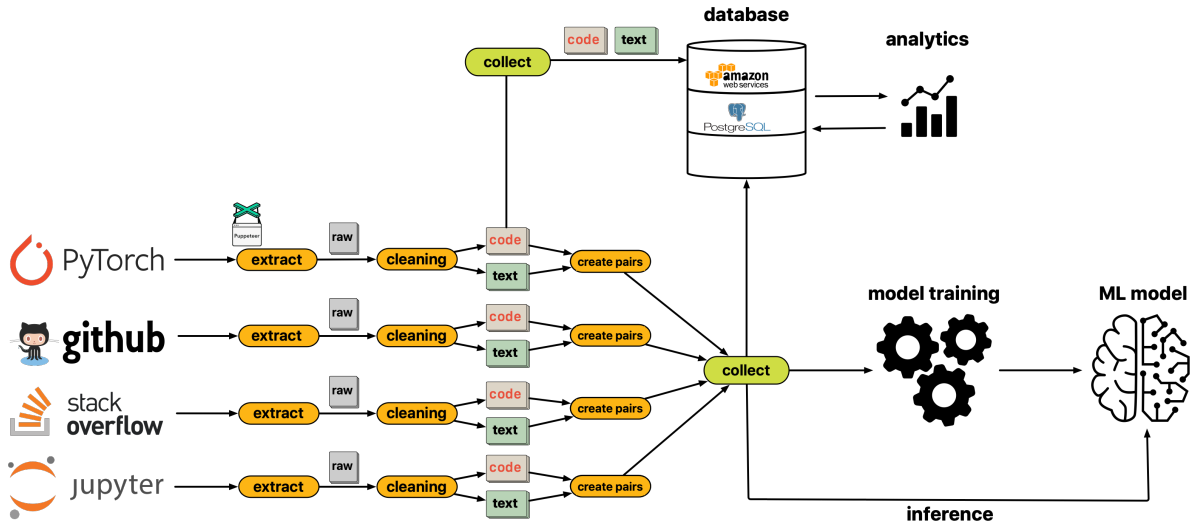

Figure 1: Overview of the data collection pipeline (best viewed in color)

solved bucket and 9,078 pairs for the unsolved bucket.

StackOverflow We extracted the StackOverflow posts with an accepted answer under the PyTorch tag via an online API called StackExchange Data Dump [19]. For each post, we paired the title and the code snippets of the accepted answer. After sampling several posts, we noticed that most of the code snippets were not pure code, but instead mixed with comments. Instead of throwing all the comments away, we thought those comments could describe the PyTorch code from another perspective. Thus, we created sub pairs in the code snippets in the accepted answer where each pair contains comments and corresponding code. This leads to another problem that which parts of code are related to the comments. To address this, we selected based on the following heuristics:

1. Docstring comments usually explain functions in Python language syntax, so those comments are paired with all code in the current function.

2. If the pound sign comments are right before code blocks like if or for statements, we paired it with all following code that have the same indentations.

3. The rest of the pound sign comments are paired with the next few lines of code until hitting a newline.

After that, we filtered the dataset based on the following criterion:

1. The code snippets must have more than 10 tokens.

2. The titles or comments must have more than 10 characters.

3. The code snippets should only contain code without any comments.

After filtering, we ended up with 8,779 StackOverflow titles/comments and code snippets pairs.

GitHub We extracted the first 1,000 repositories from the search page on GitHub by searching "PyTorch". While there were multiple files in each repository, we only looked at Python files which include "import torch". GitHub Python files are different from other sources in that code blocks are not divided into cells, and the main source of natural language is inline comments. We define a code block as code that starts immediately after a comment and ends either when it reaches the end of the current indentation or another comment. The other common phenomenon in Python files is that there can be nested comment and code pairs, such that another pair appears within the current code block which is paired with some NL. Therefore, our comment and code pairing is hierarchical such that nested NL and code pairs can be made into two pairs. While many inline comments were too short to be descriptive, we filtered out the NL and code pairs when the NL had fewer than 10 tokens and finally obtained 29,114 pairs.

JuICe Although JuICe was unique in that the dataset was preprocessed, it was critical to ensure that all data we used complemented our other sources. In the dataset, JuICe selected all natural language cells which were immediately followed by a code cell, adding a "context" of the preceding cells. We stripped away the context, preserving just the natural language and code cell pairings, narrowing down our selection to pairings which originated from notebooks that imported PyTorch. In addition, we adopt a hierarchical structure where if inline comments of greater than 10 tokens exist within the code cell, we created a child pairing of the inline comment and code following the comment. This yielded 14,044 raw pairs. We rated the pairs by their likelihood of containing actual torch code, with the strongest 4,621 pairs having the string `torch.` in them.

To make our data as clean as possible, we anonymized all data entries, preserving only natural language, code, and source type for analysis. We maintain anonymized references to the original posts, notebooks, or files to maintain a paper trail for replicability but are not linked to the dataset or model and will likely be purged before publication.

3 NLP`PyTorch` Dataset

We introduce NLP`PyTorch`: a medium-sized dataset containing matched natural language (NL) descriptions and `PyTorch` source code. The dataset contains a total of 68,398 matched NL-Code pairs collected

from the four sources as described in the previous sections. A visual representation of the matched NL-Code pairs can be seen in Fig.2. The full dataset schema can be described as:

Dataset	
Natural Language Descriptions	Programming Language (Code)

How do I create a custom neural net with two convolutions, a dropout and two fully connected layers for multi-class image classification?	`class MyNet(nn.Module):`
	`  def __init__(self):`
	`  super(MyNet, self).__init__()`
	`  self.conv1 = nn.Conv2d(1, 32, 3, 1)`
	`  self.conv2 = nn.Conv2d(32, 64, 3, 1)`
	`  self.dropout = nn.Dropout2d(0.5)`
	`  self.fc1 = nn.Linear(9216, 128)`
	`  self.fc2 = nn.Linear(128, 10)`

...	...

How do I train on a fraction of a dataset?	`l1_reg = 0`
	`for name, W in net.named_parameters():`
	`  if 'weight' in name:`
	`  l1_reg += W.norm(1)`

Figure 2: A visual of the matched NL ↔ Code pairs sampled from the NLPyTorch dataset.

In Table.1 we provide some of the aggregate summary statistics of our dataset.

Dataset Source	Total Pairs	Average NL Tokens Length	Average Code Tokens Length
github	29,114	32.27	77.42
jupyter	14,044	48.63	77.80
pytorch forum	12,851	12.42	125.93
stackoverflow	8,779	14.00	84.26

Table 1: Summary statistics of the NLPyTorch Dataset

Another way of looking at our dataset is through the box-whisker diagram given in Fig.4.

3.1 Dataset Cleaning

Before using our dataset for modeling purposes, we attempted to improve the distribution of the dataset. This is needed to prevent source bias and ensure that our dataset is relatively uniform so that training will be more predictable. At the token level, Fig. 3 and Fig. 4 reveal that each data source has its own distribution of tokens, with Jupyter and Github having long tails for NL tokens and PyTorch having an extremely large inter-quartile range for code tokens. To address this, we become more selective in our pairings, only keeping pairings with greater than 13 tokens and fewer than 500 tokens. This stabilizes our distributions, removing a large number of outliers which have a detrimental effect on model training due to the extremely long sequence of tokens which increases variance and modality of our dataset between the different collection sources.

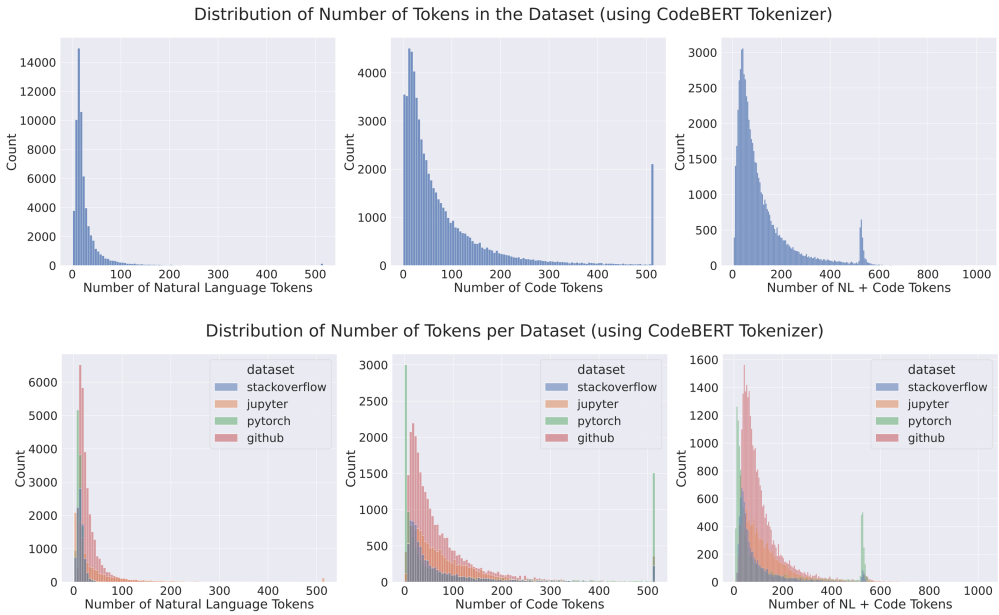

Figure 3: Histogram of the distribution of number of tokens per dataset (best viewed in color). You can see the right skew ubiquitous in all of the datasets as well as some of the outliers

Another cleaning procedure conducted for modeling was *re-formatting* so that tokenization according to each model could occur properly. For example, CodeBERT's tokenizer expects code to be split around symbols like '.', '=', '*', etc. In order to do this and preprocess for CODEnn, we utilized astor, an abstract syntax tree (AST) parser for Python, to properly parse our raw code sections. Using this, we were able to recognize symbols before rejoining with spaces between each token or symbol. We also paid attention to each model inputs and matched what each respective paper did in their processing of their datasets before they kicked off training.

4 Model Selection and Fine-tuning

There exist many pre-trained language models in code search domain. We experimented on the well-known code generation/search model named CodeBERT, which is a public pre-trained model for programming language and natural language published by Microsoft [12]. This model matches our purpose as it supports Python language and provides context embeddings for both natural languages and code. We fine-tuned the CodeBERT model parameters and trained it with different settings as suggested in the paper to get the best result. Besides CodeBERT, we chose another model called Code-Description Embedding Neural Network (CODEnn) as our baseline model. CODEnn takes advantage of embedding

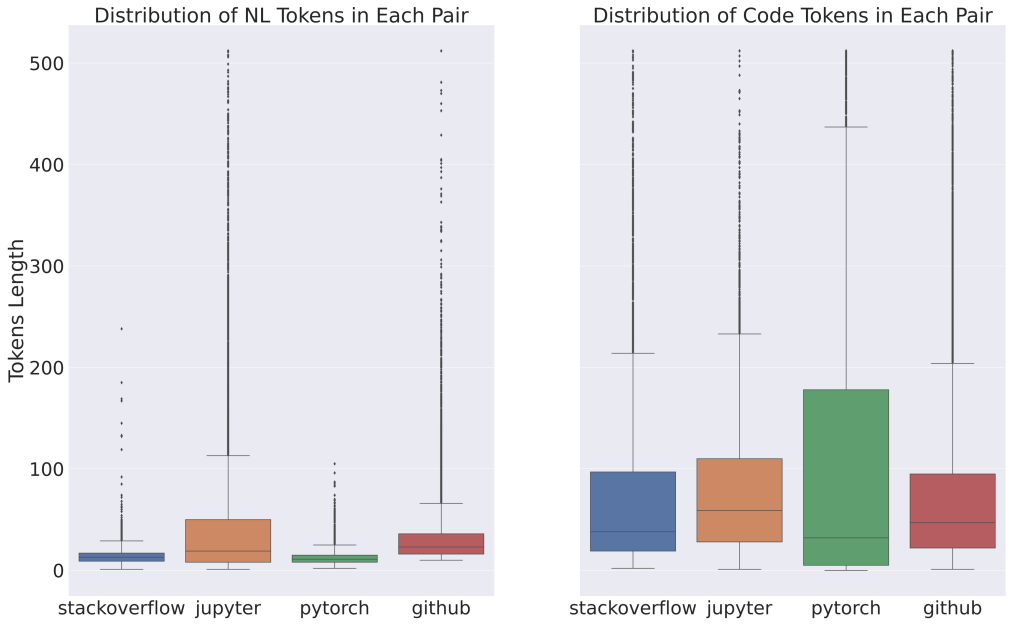

Figure 4: Box plots describing the token distributions per dataset.

code and natural language queries into a high-dimensional vector space and then calculates similarity metrics as vector distances to approximate semantic correlation between code and the query [8]. Since CODEnn is not a pre-trained model and only supports Java, we referenced the model structure and built a new model upon it to fit our dataset.

4.1 Training a Baseline Model

The CODEnn model divides the code-NL pairs into four part before training: method name, API sequence, tokens, and description[8]. In order to apply our dataset onto this model, we made following modifications to training data:

1. As mentioned in the paper, their training corpus is built upon Java methods that have documentation comments. However, in our case, most code blocks do not define Python methods. Thus, instead of extracting the method names, we used subword tokenization [20], which splits the rare words into meaningful subwords and keep the frequently used words as origin.

2. For the API sequences, we followed the ASDL Transition System used in the tranX model to grab

the syntax tree of Python code[11]. Then, we traversed through the tree and found all function calls and the corresponding calling objects.

3. For the tokens, we used the same syntax tree to tokenize the code body without punctuation. We removed the tokens that already exists in API sequences and the duplicate tokens. We also filtered out the 20 most frequently appearing tokens as most of them are Python keywords like torch, True, False, etc and thus are not representative tokens.

Fig. 5 shows an example of extracting code parts from a code snippet.

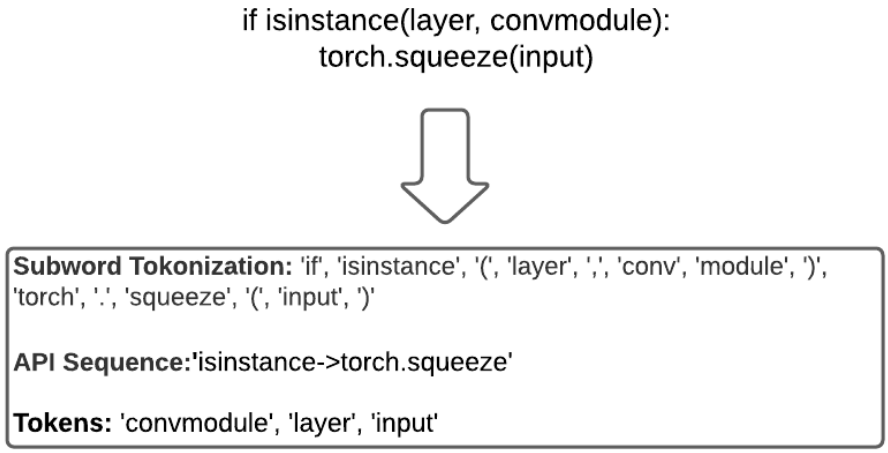

Figure 5: An example of extracting code parts given a code snippet

After collecting and preprocessing our training corpus, we were ready to feed it into the CODEnn model. The model consists of two embedding networks, CoNN and DeNN for code and natural language descriptions (NL) respectively. CoNN maps the sequences of code into a vector space, and DeNN maps the sequences of NL into the same vector space. After code and NL lie in the same vector space, cosine similarity can be used to measure the matching of them. DeNN is a single LSTM network, and CoNN consists of several networks.

- One LSTM for subword tokenization
- One LSTM for API sequences
- One MLP for tokens
- One feed-forward network to fuse the three embeddings

For each positive sample, $< C, NL^{+} >$ (a pair of code and NL), we randomly selected another NL to generate negative samples, $< C, NL^{-} >$, to keep the number of positive and negative samples balanced. The loss function is defined to maximize the similarity of the embeddings of code and NL in positive samples and minimize it in negative samples.

$$\sum_{<C_{emb}, NL^{+}_{emb}, NL^{-}_{emb}>} max(0, \epsilon - \cos(C_{emb}, NL^{+}_{emb}) + \cos(C_{emb}, NL^{-}_{emb})) \tag{1}$$

The accuracy is defined as follows: For a given NL, we count 1 if the target code is within the top K (e.g. 10) codes returned by the model. We trained the baseline model for 15 epochs and achieved test accuracy 2.67% and training accuracy 11.45%.

4.2 Fine-tuning the CodeBERT Model

Pre-training Transformer [21] based models [22, 23, 24] with their ability to achieve state-of-the-art performance on a wide variety of NLP tasks (e.g. machine translation, sequence classification etc.) also inspired the neural programming systems (PS) and programming languages (PL) community. Transformer based models pre-trained on paired code and natural language datasets achieved state-of-the-art results in tasks such as: code search, code translation, code generation, code documentation [12, 25, 26]. We selected CodeBERT as to fine-tune because of two main reasons: learned representations for python and its pre-training with a hybrid objective function which supports code search.

CodeBERT follows BERT [22] and RoBERTa [23] for pre-training recipes of the ubiquitous Transformer architecture [21]. It uses two pre-training objectives: Masked Language Modeling (MLM) and Replaced Token Detection (RTD). MLM objective is trained on *bimodal* data which consists of natural language - code (NL-PL) pairs (w_i, c_i) and RTD objective [27] is trained on *unimodal* data which consists of just code tokens or just natural language words as inputs. Similarly to BERT, MLM is what allows the CodeBERT model to learn bi-directional representations for both code and natural language and RTD is what allows the model to learn to discriminate whether a code token or word is the corrupt ("fictitious") (0) or not (1). We follow a similar procedure for fine-tuning CodeBERT on our dataset as it was described in the CodeBERT paper. Our adjusted fine-tuning steps are listed below:

1. Grab a pre-trained CodeBERT model (we used `microsoft/codebert-base`)
2. Extend the dataset to create a balanced negative $label = 1$ and positive $label = 0$ samples as shown in Fig.6
    - For each record in our dataset which consists of NL-PL pairs (w_i, c_i)

- For about half the records, strip the code and replace it with another code uniformly sampled from records that belong to same source.
- For the remaining half, strip natural language explanation and replace it with another NL explanation uniformly sampled from records that belong to same source.

3. Create train/test/validation splits (80/10/10) respectively and independently tokenize both NL and Code pairs, stitch them to form input to the model.

4. Use the final hidden vector $C \in \mathbb{R}^H$ for the $[CLS]$ token and push it through a fully connected layer followed by a softmax function and compute the binary cross entropy loss.

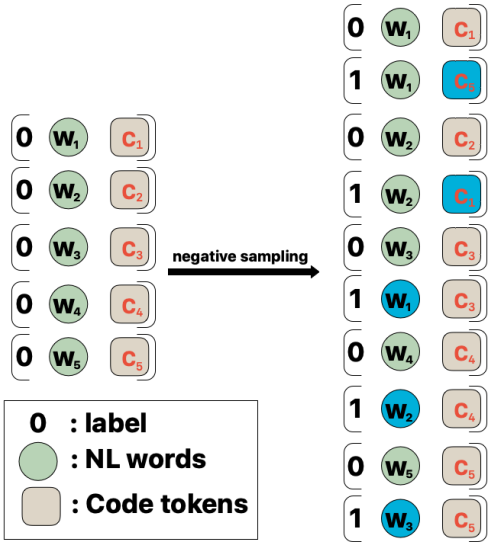

Figure 6: A visual representation of the negative sampling process that generates unmatched pairs using original pairs in our dataset with replaced word or code tokens colored in blue.

We train for a total of 4 epochs, using a single Nvidia V100 GPU with training and evaluation batch sizes of 32, a maximum sequence length of 200 and a starting learning rate of $2e-5$ with a linear warm-up schedule. After fine-tuning is done we can make predictions on a candidate NL-Code pair by passing the input through the learned model and use the `[CLS]` token representations to pass it through e fully connected layer followed by softmax to get the semantic relevance between the NL and Code pair. The problem of code-search then simply becomes finding such Code pair that achieves the maximum probability score among a candidates as shown in Fig. 7.

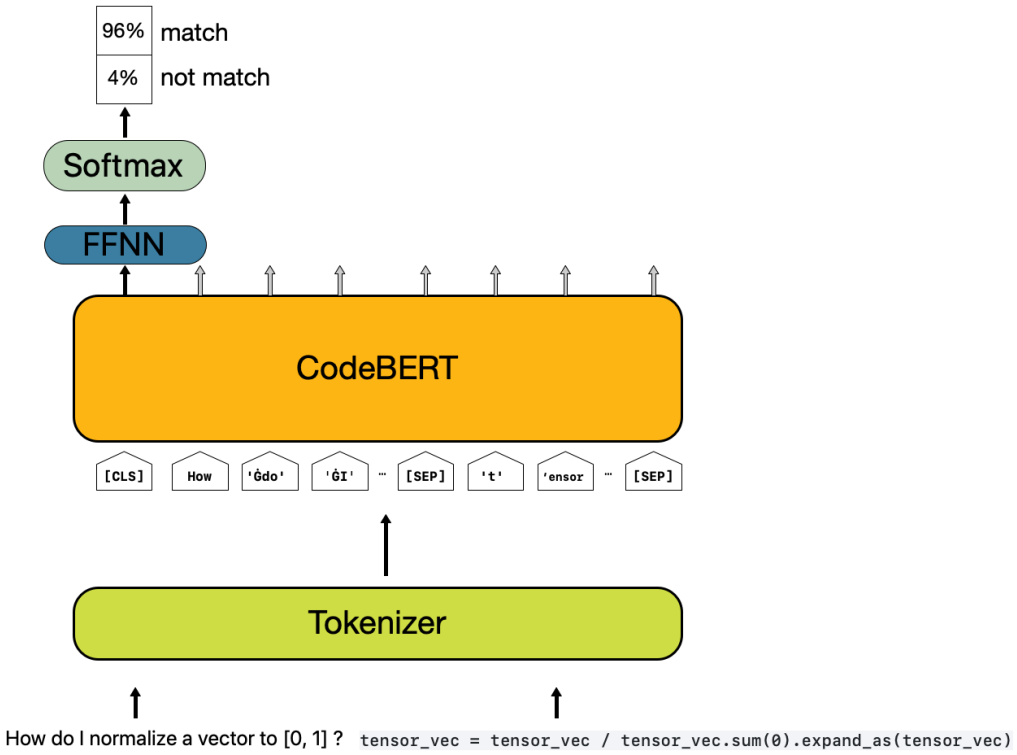

Figure 7: A visual representation of code search at inference time using the fine-tuned CodeBERT model on NLPyTorch dataset.

5 Conclusions and Future Work

In this paper, we proposed a novel dataset called NL`PyTorch`, which contains matched natural language (NL) descriptions and `PyTorch` source code. We extracted the data from four sources, PyTorch Discussion Forum, StackOverflow, GitHub and JuICe, and curated them based on the source format. In consideration of a uniform distribution among four sources, we further cleaned the dataset by removing the outliers and reformatting the tokens, and finally gained 55247 pairs. After building the dataset, we applied it on two well-known code search models, CODEnn and CodeBERT. We used CODEnn as our baseline model and modified the preprocessing step and model structure to better fit our dataset. CODEnn only supports the Java languages and we need to train the embedding networks from scratch. As we only had limited number of code-NL pairs, the performance is not satisfactory, so we took advantage of the pretrained model, CodeBert. We extended the dataset to create a balanced negative and positive samples and fine-tuned the model.

In the future, we will investigate more on standardizing the source code structure so that code search models can better understand the semantics meaning and provide more effective representations. Also we will try our dataset on more advanced model like GraphCodeBERT, which came out a couple of months ago. GraphCodeBert pre-trains on not only the NL-Code pairs similar to what was used in CodeBERT but also on a variable flow graph which captures more of the inherent structure present in code.

Acknowledgements

We thank our advisor Alvin Cheung for his guidance, support and discussions which resulted in us producing this dataset and the experimental results especially in these trying times. We also thank our colleagues at Berkeley Programming Systems Research group for all the fruitful discussions. This work was accomplished fully remote and to this day authors have not yet met in real life. They hope to do so to grab a beer once it is fully safe.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[2] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of machine learning research*, vol. 12, no. ARTICLE, pp. 2493–2537, 2011.

[3] J. Zou, M. Huss, A. Abid, P. Mohammadi, A. Torkamani, and A. Telenti, "A primer on deep learning in genomics," *Nature genetics*, vol. 51, no. 1, pp. 12–18, 2019.

[4] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[5] P. Brusilovsky, "Webex: Learning from examples in a programming course." in *WebNet*, vol. 1, 2001, pp. 124–129.

[6] R. Agashe, S. Iyer, and L. Zettlemoyer, "Juice: A large scale distantly supervised dataset for open domain context-based code generation," 2019.

[7] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[8] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[9] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1139–1149. [Online]. Available: https://www.aclweb.org/anthology/P17-1105

[10] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers," *CoRR*, vol. abs/1911.04942, 2019.

[11] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," *CoRR*, vol. abs/1810.02720, 2018.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020.

[13] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[14] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *CoRR*, vol. abs/1809.08887, 2018.

[15] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, vol. abs/1709.00103, 2017.

[16] Y. Roh, G. Heo, and S. E. Whang, "A survey on data collection for machine learning: a big data – ai integration perspective," 2019.

[17] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," *CoRR*, vol. abs/1905.03813, 2019.

[18] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3173606

[19] "Stack exchange data dump : Stack exchange, inc. : Free download, borrow, and streaming," online, March 2021, accessed 15 March 2021. [Online]. Available: https://archive.org/details/stackexchange

[20] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[22] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[23] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[24] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[25] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *CoRR*, vol. abs/2009.08366, 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[26] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[27] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020.