CS230

Making Neural Architectures Learn to Compile (Natural Language Processing)

Jeffrey Shen
jshen2@stanford.edu

Abstract

Neural architectures can, with relative difficulty, learn to execute simple programs, e.g. addition or sorting. However, if the tasks for memory, arithmetic, and branching are abstracted away into operations, it is not well studied whether neural architectures can convert from higher-level operations to them. We train a transformer model to compile from Python to Python bytecode. We first train using unsupervised methods, and find that parallel data even without supervision can dramatically improve the performance. We then train using supervision, but despite further improving the performance, the transformer is still unable to compile well because certain tokens are dependent on later tokens in the sequence. We attempt to allow the transformer to predict tokens out of order, but encounter issues with the training setup, and propose a solution.

1 Introduction

Several studies [12, 9, 3, 2] have shown that it is possible for a neural architecture to learn to execute various programs. However, the learned tasks are usually very simple, e.g. addition or sorting. We note that even simple arithmetic tasks are relatively difficult for neural architectures to learn.

In this work, we investigate whether neural architectures can learn to compile. For a computer to execute a program written in a high-level language, usually the source code is first translated to a low-language language before it is executed on hardware. By abstracting away the implementation of the low-level language, we can train neural architectures to convert from higher-level abstractions into lower-level abstractions, without needing to learn basic operations for memory, arithmetic, and branching. Furthermore, compared to, say, translating from natural language to source code [1], we can more easily judge the performance of a neural compiler, since we already have well-established baselines, i.e. standard (non-neural) compilers for each language.

The input for our model is Python source code, which we train to output human-readable Python bytecode which can be generated by a disassembler module contained in the Python standard library[1]. We follow the approach outlined by Lachaux et al. [6] which uses a transformer architecture [11].

2 Related work

A number of studies investigate execution of relatively simple programs [12, 9, 3, 2]. Zaremba and Sutskever [12] look at producing an integer output from generated Python source code of various lengths. Reed and de Freitas [9] propose a neural architecture that is able to learn various programs, like addition and sorting, using only a scratch pads, by training on execution traces.

[1] https://docs.python.org/3/library/dis.html

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

Various studies consider translation between different programming language pairs. Fu et al. [5] study neural decompilation from a lower-level programming language to a higher-level one. Barone and Sennrich [1] translate between a programming language, i.e. Python, and its natural language description, i.e. documentation strings.

Lachaux et al. [6] introduce a neural transcompiler that can translate functions between C++, Java, and Python. Their approach is based on the methods of Conneau and Lample [4], which enables them to train in an unsupervised way on monolingual source code gathered from various Github repositories. Our work uses the approach and released source code of Lachaux et al. [6].

3 Dataset and Features

As in [6], we build our dataset using source code from a GitHub public dataset via Google BigQuery[2]. We choose to download Python files from repositories with an MIT license or public domain license. Since the dataset is fairly large, we shard the dataset into 128 shards based on the MD5 hash of the source code, and use only files in shard 0. This gives us a dataset with 32081 files, totaling 196MB.

We apply the `dis` function from `dis`, the disassembler Python module, to generate bytecode. The module is tied to the Python interpreter and version used to compile the code, and in particular, we use Python 3.7.9, since versions prior to 3.7 do not recurse into all functions. Any Python code that did not compile, including, in particular, Python 2 code, is excluded. In Python, files can be independently compiled to bytecode without compiling dependencies. We give an example of the output of `dis` in Figure 1.

We use the Python tokenizer from [6] to remove unnecessary whitespace. There is no such tokenizer for the `dis` output, so we build our own tokenizer. BPE codes [10] are learned on the tokens using the fastBPE implementation[3] and applied to the tokenized files before being passed to the model.

Python function	`dis` output
	1 0 LOAD_CONST 0 (<code object fib at 0x7fed0bcc5540, file "<dis>", line 1>) 2 LOAD_CONST 1 ('fib') 4 MAKE_FUNCTION 0 6 STORE_NAME 0 (fib) 8 LOAD_CONST 2 (None) 10 RETURN_VALUE Disassembly of <code object fib at 0x7fed0bcc5540, file "<dis>", line 1>:
	2 0 LOAD_FAST 0 (n) 2 LOAD_CONST 1 (1) 4 COMPARE_OP 1 (<=)
`def fib(n):` `  if (n <= 1):`	6 POP_JUMP_IF_FALSE 12
`  return n`	3 8 LOAD_FAST 0 (n) 10 RETURN_VALUE
`return fib(n - 1) + fib(n - 2)`	4 >> 12 LOAD_GLOBAL 0 (fib) 14 LOAD_FAST 0 (n) 16 LOAD_CONST 1 (1) 18 BINARY_SUBTRACT 20 CALL_FUNCTION 1 22 LOAD_GLOBAL 0 (fib) 24 LOAD_FAST 0 (n) 26 LOAD_CONST 2 (2) 28 BINARY_SUBTRACT 30 CALL_FUNCTION 1 32 BINARY_ADD 34 RETURN_VALUE

Figure 1: A Python function implementing fibonacci, and the human-readable disassembled Python bytecode generated using `dis`. The `dis` output has 8 columns, including, in order, the source code line number, current instruction indicator (not shown), jump target marker ("»"), instruction offset, instruction name, instruction argument, and the argument's python representation. The start of the bytecode for the `fib` function is indicated by the line starting with "Disassembly of."

[2]https://console.cloud.google.com/marketplace/details/github/github-repos

[3]https://github.com/glample/fastBPE

3.1 Evaluation dataset

For evaluation, we use a GeeksForGeeks[4] parallel dataset released by Lachaux et al. [6], which contains over 800 standalone functions in C++, Java, and Python, split into validation and test examples. We generate bytecode from the Python functions and extract the bytecode function starting with the line "Disassembly of [...]." We only take the paired examples where the bytecode has a single function. Notably, this avoids examples with list comprehensions and closures, which generate additional functions. For faster validation, during training, we only used up to 20 validation examples.

3.2 Baseline dataset

For the baseline dataset, we use the entire source code and generated bytecode for pretraining, and we extract standalone functions only, including comments, for the training phase. To follow a similar approach as [6], we first generate bytecode from Python files and then extract standalone functions directly from the bytecode of the whole files. For pretraining, the dataset was split into 8 parts (intended for distributed training) of which we mistakenly only used one, but for training we used the whole dataset.

3.3 Parallel dataset

We also constructed a parallel dataset that more closely resembles the evaluation dataset. Instead of extracting functions from whole bytecode files, we first extract the standalone Python functions, generate bytecode for each function, and then extract the bytecode function, keeping only the paired examples where there is a single bytecode function.

4 Methods

We leveraged the XLM model [4] implementation used by Lachaux et al. [6] to build TransCoder. Each model is a transformer [11] with 6 layers, 8 attention heads, and 1024 hidden units, implemented in PyTorch [8]. Note that the encoder takes vectors of tokens, positions, and a language representing a source language sequence and outputs an encoding. The decoder receives the encoding and a target language sequence, and it outputs next-token probabilities.

4.1 Unsupervised methods

We use three different objectives for unsupervised pretraining and training, with the masked language modeling (MLM) objective for pretraining, followed by the denoising auto-encoding (AE) objective and back-translation (BT) objective for training.

In MLM, tokens are masked at random, and the model is asked to recover the original tokens. Only the encoder is trained, using an additional linear-softmax classifier that predicts the masked-token. In AE, tokens are masked, deleted, and shuffled, and the whole encoder-decoder model is asked to translate from the mutated sequence to the original sequence in the same language. Finally in BT, the model first translates from one language to another and learns to recover the original code from the output. In this way, the model learns to translate in an unsupervised way without parallel data.

We needed to tune some of the batch sizes and epoch sizes to train on only one, smaller GPU. In the pretraining phase, we used a batch size of 8, and epoch size of 5000 steps. For the training phase, we grouped examples into 1000 tokens per batch, and an epoch size of 1500 steps.

4.2 Supervised methods

The XLM model also allows a mode for supervised machine translation (MT). In MT, the model is given parallel examples in each language, and the whole encoder-decoder model is trained to translate from a source language sentence to the target language sentence. In contrast with transcompilation, we can easily generate and leverage parallel datasets between Python and Python bytecode.

[4] https://practice.geeksforgeeks.org/

4.3 Any token prediction

Instead of translating each token in order, we can also try to allow the decoder to predict tokens out of order. The output layer of the decoder has a single linear-softmax classifier to predict next-token probabilities. We modify the XLM decoder to train an additional linear-softmax classifier to predict a position. If y is a vector of target token indices, and p be a vector of target positions such that $\{(y_i, p_i)\}$ is the set of yet to be predicted tokens, we then compute the loss for a probability vector $\hat{y}$ and position probability vector $\hat{p}$ as:

$$L(y, p, \hat{y}, \hat{p}) = -\log \sum_i \hat{y}_{y_i} \hat{p}_{p_i}. \tag{1}$$

That is, we set the loss to the negative log likelihood that the decoder predicts any one of the yet to be predicted tokens (y_i, p_i). We call this the any token prediction (ATP) objective.

The model already incorporates (learned) input embeddings for both the token and position, so we can already feed the decoder input sequences out of order. We first train the model with the ATP objective given the target language sequence in order. We also train the model with the ATP objective on permutations of the sequences of token-position pairs.

5 Results

We first report results on three different models we trained to completion: an XLM model using unsupervised methods on the baseline dataset (XLM-BU), using unsupervised methods on the parallel dataset (XLM-PU), and using supervised methods on the parallel dataset (XLM-PS). The XLM-PS model was initialized from XLM-PU model weights. We also partially trained two XLM models with the ATP objective, the first on parallel datasets with target sequences given in order (XLM-PO) and the second with target sequences permuted (XLM-PP). XLM-PO was initialized using XLM-PS and XLM-PP using XLM-PO. We note that all models are also capable of performing decompilation (`dis` to Python) in addition to compilation.

We report three metrics in Table 1 for each completed model and both translation directions: reference match, BLEU score [7], and computational accuracy. Reference match is the percentage of translations that match the reference exactly. In the case of `dis` outputs, however, we ignore any memory addresses (e.g. `0x7fed0bcc554`) when matching against the reference. Lachaux et al. [6] introduced a new metric called computational accuracy. Their GeeksForGeeks test set includes a suite of auto-generated unit tests, and we report the percentage of translations that passed their unit tests. To test `dis` translations, we compile the Python unit test and the reference Python function, and swap out the reference code object with the bytecode specified by the `dis` translation.

Table 1: Results for completely trained models (XLM-BU, XLM-PU, XLM-PS). The best model for Python $\rightarrow$ `dis` (XLM-PS) only produces bytecode that passes unit tests 18.2% of the time, despite having a BLEU score of 81.9.

Python $\rightarrow$ `dis`	XLM-BU	XLM-PU	XLM-PS
Reference Match	0.0	0.0	13.7
BLEU	19.6	35.5	81.9
Computational Accuracy	0.0	0.4	18.2
`dis` $\rightarrow$ Python	XLM-BU	XLM-PU	XLM-PS
Reference Match	0.0	0.4	12.4
BLEU	4.9	13.1	82.1
Computational Accuracy	0.6	1.2	47.5

We note that compilation to Python bytecode is actually a rather unforgiving problem for the XLM models. Despite having a high BLEU score of 81.9, the best model only had a computational accuracy of 18.2%. For reference, the worst language pair for the TransCoder model [6] was Python to Java, with a worse BLEU score of 64.6 but computational accuracy of 24.7%. Furthermore, their model is unsupervised, and uses a non-parallel dataset, similar to XLM-BU. Even decompilation is easier, since XLM-BU is able to produce correct Python programs without exactly matching the reference output, whereas doing so with Python bytecode is rare.

The issue is best illustrated in Figure 2. For the SETUP_LOOP instruction, the model must output the loop length and the loop end. However, it must do so before writing the loop itself, which is very difficult. 82.6% of the validation set (and likely the test set) contains a SETUP_LOOP instruction.

Python function	XLM-PS output (function body)	`dis` output (function body)

`def linearSearch(arr, n):`
`  for i in range(n):`
`  if arr[i] is i:`
`  return i`
`  return -1`
	2 0 SETUP_LOOP 36 (to 38)
	2 LOAD_GLOBAL 0 (range)
	4 LOAD_FAST 1 (n)
	6 CALL_FUNCTION 1
	8 GET_ITER
	>> 10 FOR_ITER 24 (to 36)
	12 STORE_FAST 2 (i)
	
	3 14 LOAD_FAST 0 (arr)
	16 LOAD_FAST 2 (i)
	18 BINARY_SUBSCR
	20 LOAD_FAST 2 (i)
	22 COMPARE_OP 8 (is)
	24 POP_JUMP_IF_FALSE 10
	
	4 26 LOAD_FAST 2 (i)
	28 RETURN_VALUE
	30 JUMP_ABSOLUTE 10
	>> 32 POP_BLOCK
	
	5 >> 34 LOAD_CONST 1 (- 1)
	36 RETURN_VALUE
		2 0 SETUP_LOOP 32 (to 34)
		2 LOAD_GLOBAL 0 (range)
		4 LOAD_FAST 1 (n)
		6 CALL_FUNCTION 1
		8 GET_ITER
		>> 10 FOR_ITER 20 (to 32)
		12 STORE_FAST 2 (i)
		
		3 14 LOAD_FAST 0 (arr)
		16 LOAD_FAST 2 (i)
		18 BINARY_SUBSCR
		20 LOAD_FAST 2 (i)
		22 COMPARE_OP 8 (is)
		24 POP_JUMP_IF_FALSE 10
		
		4 26 LOAD_FAST 2 (i)
		28 RETURN_VALUE
		30 JUMP_ABSOLUTE 10
		>> 32 POP_BLOCK
		
		5 >> 34 LOAD_CONST 1 (- 1)
		36 RETURN_VALUE

Figure 2: A validation example where the XLM-PS model failed to produce a valid bytecode output. The model is unable to predict the jump target for the SETUP_LOOP instruction in the first line.

5.1 Any token prediction

If the XLM model could produce the tokens in a different order, it could learn to predict the SETUP_LOOP argument after writing the rest of the loop. We trained XLM-PO, which receives tokens in order, until it was able to predict any one of the remaining token-position pairs with an accuracy on the validation dataset of over 81%. Eventually, we would expect XLM-PO to match or exceed the per-token accuracy of XLM-PS, which was around 96%, because of the similarity of the tasks. However, we paused training to focus on predicting tokens from permuted sequences to force the model to also learn to predict tokens in different orders. However, after training on permuted sequences, the accuracy of XLM-PP predicting any of the next tokens for an in order sequence dropped to 10% on the validation dataset. We postulate that on average, XLM-PP is receiving an easier task - identifying any word in a randomly masked passage is much easier than identifying one of the next words in a passage where the second half is masked. In either case, manual inspection of the outputs revealed that both models were predicting a NEW_LINE token a vast majority of the time and not learning to predict the more salient parts of the output sequence.

6 Conclusion

The compilation task as presented is difficult, even for state of the art neural architectures, to learn. Aligning the distributions of the languages in the training data with each other, even without supervision, and supervised training using parallel data drastically improve the performance of the model. However, even the best model is unable to predict the future. For a fairer evaluation of whether neural architectures can compile to lower-level abstractions, we could change the bytecode language, for example, by moving all jump instruction arguments to the end of the output. Given the performance on the rest of the dataset, it's possible that such a trivial change could let the model reach near perfect computational accuracy.

Still, the question of whether we can make a transformer model more robust to the ordering of the target sequence by extending the available output actions is interesting. In doing so, however, it is important that the outputs feed back into the inputs. Instead of using a predetermined sequence, we could use the output of the ATP decoder to determine the next token to be passed as input. Even more generally, we could see the decoder as outputting an action to apply to an environment, i.e. the target sequence, and translate translation to a reinforcement learning problem. Lastly, we would like to see if these techniques can generalize to other problems in NLP.

References

[1] A. V. M. Barone and R. Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *CoRR*, abs/1707.02275, 2017. URL http://arxiv.org/abs/1707.02275.

[2] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. Torr. Adaptive neural compilation. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1444–1452. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6411-adaptive-neural-compilation.pdf.

[3] J. Cai, R. Shin, and D. Song. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=BkbY4psgg.

[4] A. Conneau and G. Lample. Cross-lingual language model pretraining. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 7059–7069. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/8928-cross-lingual-language-model-pretraining.pdf.

[5] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3708–3719. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/8628-coda-an-end-to-end-neural-program-decompiler.pdf.

[6] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample. Unsupervised translation of programming languages, 2020.

[7] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://www.aclweb.org/anthology/P02-1040.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[9] S. E. Reed and N. de Freitas. Neural programmer-interpreters. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06279.

[10] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015. URL http://arxiv.org/abs/1508.07909.

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

[12] W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL http://arxiv.org/abs/1410.4615.