C3PO: A Lightweight Copying Mechanism for Translating Pseudocode to Code

Vishruth Veerendranath and **Vibha Masti** and **Prajwal Anagani** and **Mamatha HR**
Department of Computer Science and Engineering, PES University, Bangalore, India
{vishruth, vibha, prajwala}@pesu.pes.edu, mamathahr@pes.edu

Abstract

Writing computer programs is a skill that remains inaccessible to most due to the barrier of programming language (PL) syntax. While large language models (LLMs) have been proposed to translate natural language pseudocode to PL code, they are costly in terms of data and compute. We propose a lightweight alternative to LLMs that exploits the property of code wherein most tokens can be simply copied from the pseudocode. We divide the problem into three phases: Copy, Generate, and Combine. In the Copy Phase, a binary classifier is employed to determine and mask the pseudocode tokens that can be directly copied into the code. In the Generate Phase, a Sequence-to-Sequence model is used to generate the masked PL code equivalent. In the Combine Phase, the generated sequence is combined with the tokens that the Copy Phase had masked. We show that our C3PO models achieve similar performance to non-C3PO models while reducing the computational cost of training as well as the vocabulary sizes.

1 Introduction

In recent years, computer programs have found applications in almost every field, from scientific to artistic fields. The demand and cost for programmers have gone up because writing code is a specialised skill. Although people may be able to describe the functionality of the required code, the syntax of a programming language serves as a barrier to writing code (Denny et al., 2011). Recently there has been an increase in Low-Code applications that only require the functionality of code to be specified as pseudocode in Natural Language (NL), which is then translated to source code in a Programming Language (PL). Pseudocode enables people unfamiliar with a PL's syntax to write the functionality of the required code in NL, and allows programmers to write PL-independent code, which emphasizes functionality over syntax.

Translating pseudocode to code is cumbersome due to the complex structures of programs that result from their syntax, semantics and logic. Existing state-of-the-art Pseudocode-to-Code translators, like Codex (Chen et al., 2021), which is used to power GitHub Copilot (GitHub, 2021), and CodeT5 (Wang et al., 2021), are being powered by complex transformer LLMs like GPT-3 (Brown et al., 2020) and T5 (Raffel et al., 2020) respectively. They are pre-trained on very large code datasets like CodeSearchNet (Husain et al., 2019) and CodeXGLUE (Lu et al., 2021). While these transformer architectures are good at generalizing to many downstream PL-related tasks, they require high-performance computational resources, large amounts of data, and significant time to train.

Our contributions are as follows: we propose C3PO (**C**omputationally efficient **C**opying mechanism for **C**onversion from **P**seudocode to c**O**de), a lightweight alternative to the current translators. We exploit the property of code wherein a large number of tokens (like identifiers and variable names) are present in both pseudocode and its corresponding PL code. These tokens can therefore be simply copied into the resultant PL code translation. The remaining tokens can then be generated based on PL syntax. We divide the task of pseudocode to code translation into three phases: the Copy Phase, Generate Phase and the Combine Phase.

In the Copy Phase, we use a Decision Tree Classifier to decide whether each token in the pseudocode needs to be copied or translated. In the Generate Phase, a Sequence-to-Sequence (Seq2Seq) model takes in a pseudocode sequence, in which the tokens that can be copied are masked, and generates a masked PL sequence. In the Combine Phase, the generated masked PL code is unmasked with the appropriate true pseudocode tokens. C3PO has been trained on the SPoC dataset (Kulal et al., 2019), which consists of human-written pseudocode lines in English with their corresponding C++ code lines.

Proceedings of the AACL-IJCNLP 2022 Student Research Workshop, pages 47–53

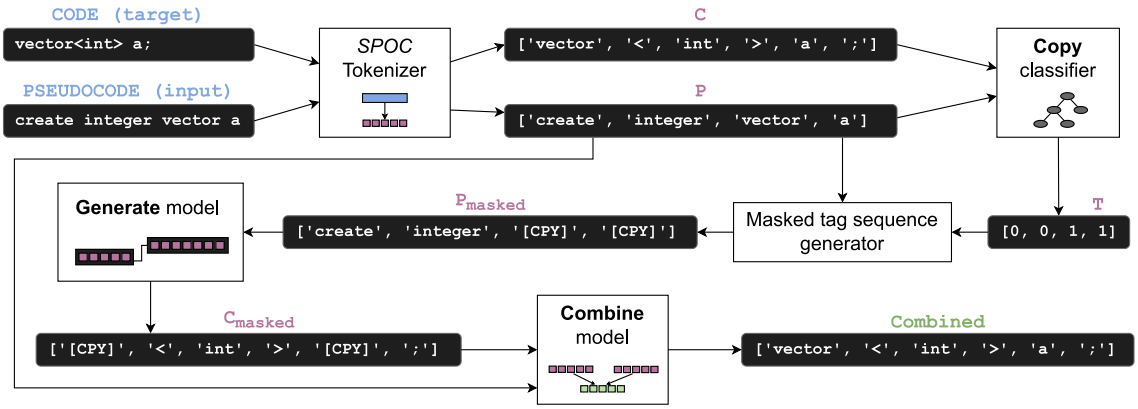

Figure 1: **An illustrative example of pseudocode being translated to C++ code during training**. The pseudocode and code are tokenized (P and C) and preprocessed together to get the truth-label for training the Copy Classifier. The binary tag sequence (T) is used to mask the tokens (P_{masked}) and generate the masked code sequence (C_{masked}). This is finally combined with the copied tokens from the input P, to result in the translation $Combined$.

2 Related Work

2.1 Pseudocode to Code translation

There have been many approaches taken to translate natural language input into a programmatic context. Some of the earlier works like Seq2SQL (Zhong et al., 2017) focused on the simpler task of generating SQL queries, where the query was segmented into its constituent parts (like `SELECT`, `WHERE`) with separate objectives for each part.

Kulal et al. (2019) approached the task as a search-based line-by-line translation task using LSTM Seq2Seq. With the introduction of large corpora for code (Lu et al., 2021), transformer architectures like Code-T5 (Wang et al., 2021), Codex (Chen et al., 2021) and CodeGen (Nijkamp et al., 2022) have been pre-trained on various objectives to generalize to various code related downstream tasks such as Code Generation and Summarization. While these facilitate fine-tuning, they require a lot of computational resources for training as well as inference.

There have also been approaches to translating code using code search techniques rather than synthesising code (Feng et al., 2020), (Neelakantan et al., 2022). These approaches generate embeddings using a variation of the BERT encoder and then use similarity metrics to search for the most semantically similar code from a corpus of code such as CodeSearchNet (Husain et al., 2019). For another such BERT model (Norouzi et al., 2021), the authors also proposed two new evaluation metrics namely *copy accuracy* and *generation accuracy*, and maximise each of these separately.

2.2 Copying Mechanism

CopyNet (Gu et al., 2016) describes a copy mechanism for natural language text, where the decoder in a Seq2Seq model is modified to probabilistically predict words from either the *copy-mode* or *generate-mode*. A related methodology called Pointer Networks (Vinyals et al., 2015) uses the attention mechanism to create pointers to the input words. The deobfuscation objective in DOBF (Roziere et al., 2021) provides an alternative to masked language modelling (MLM) for identifier names during pre-training.

3 Problem Definition

As we handle the task on a line-by-line basis, the task can be formulated as follows: The input sentence consists of a sequence of m pseudocode tokens $P = P_1, P_2 ... P_m$, and the objective is to translate this into its corresponding sequence of n code tokens $C = C_1, C_2, ... C_n$. We additionally split the pseudocode tokens P into 2 sets of tokens — P_{cpy} and P_{gen} — where $P = P_{cpy} \cup P_{gen}$.

The main idea for the copying mechanism stems from the fact that a good number of the code tokens (identifier names, constants and keywords) in the code translation are also present in the pseudocode input. Since such tokens can be simply copied into the translation, generating them from a Seq2Seq model is not necessary. Such tokens are referred to as the set of *copied tokens* denoted by P_{cpy}. In the SPoC dataset, we observed that the mean ratio of copied tokens to sequence length in pseudocode was around 60%.

Pseudocode	Ground-Truth Code	Generated Code
`if x is even` `n,b = integers with b=0`	`if(x % 2 == 0)` `int n, b=0;`	`if(x % 2 == 0)` `int n, b=0;`
`while read n` `print "NO"`	`while(cin >> n)` `cout << "NO" << endl;`	`while n cin >> ;` `cout << "NO" << "\n";`

Table 1: Examples of pseudocode, corresponding true code and generated code (correct in green, wrong in red)

The tokens that are not common to the pseudocode and code, referred to as the set of *generated tokens* and denoted by P_{gen}, describe code functionality in natural language and would have to be generated by a Seq2Seq model. For instance the token `read` will correspond to `cin` in C++.

4 Methodology

We devise a three-stage solution, which we call *C3PO*, to the above-defined problem. The three phases – *Copy*, *Generate* and *Combine* – are defined in an attempt to reduce the complexity of the model. The stages of *C3PO* are illustrated with an example in Fig. 1.

4.1 Copy Phase

We use a *Binary Classifier* to determine which tokens in the pseudocode are present in the code. For each token P_i in the pseudocode input, the binary classifier would discriminate which type of token it is – whether the token can be copied into the code output ($P_i \in P_{cpy}$), or whether it would have to be generated by the Seq2Seq model ($P_i \in P_{gen}$).

The binary classifier acts as a *tagger* on the pseudocode, generating a tag sequence. We define two different representations for the tag sequence. The first, named *binary tag sequence* (T) is a binary array – 1s referring to copied tokens, and 0s referring to generated tokens. An example of the binary tag sequence is shown in the output of the Copy classifier in Fig. 1.

The second, named *masked tag sequence* (P_{masked}) is a processed form of the input pseudocode sequence P. To facilitate this masking, we assign a special token which we call the *Copy Mask Token*, represented as `[CPY]`. We mask all tokens that belong to set P_{cpy} by replacing such tokens with the `[CPY]` tag, before passing the sequence as input to the Seq2Seq model.

4.2 Generate Phase

The masked tag sequence (P_{masked}) is provided as input to a Seq2Seq model, which would generate the corresponding masked code output (C_{masked}). C_{masked} is also masked with `[CPY]` tags when the output corresponds to a token that was originally masked in the pseudocode. The Seq2Seq model would only influence the position of copied tokens in the output code, and not the token itself.

4.3 Combine Phase

The generated masked code output (C_{masked}) needs to be transformed into actual code (C). To simplify our model, we assume that the order in which the masked tokens appear in the true code is the same as the order in which they appear in the final code, although this might not be the case at all times. This is a fair assumption to make in most cases as shown in Section 7.1. We replace only the `[CPY]` tags in C_{masked} directly with the copied tokens in set P_{cpy} in their order of occurrence in P. This simple combination of the two previous two phases leads to the resultant translation.

5 Models and Experiments

5.1 Dataset

The SPoC dataset (Kulal et al., 2019) is used for our experiments. It contains 18,356 C++ programs and their human-authored pseudo-code in English language. The dataset covers a wide variety of programs with multiple programs for a single problem statement sourced from CodeForces contests.

5.2 Copy Model

For the copy phase, a decision tree is used to predict whether or not each token gets directly copied into the code. The decision tree was created using sklearn's (Buitinck et al., 2013) `DecisionTreeClassifier` API and was trained on *pseudocode (input)* sequence from the entire SPoC training dataset, as shown in Fig. 1. For every token in every sentence, the following explicit features were passed as input to the tree:

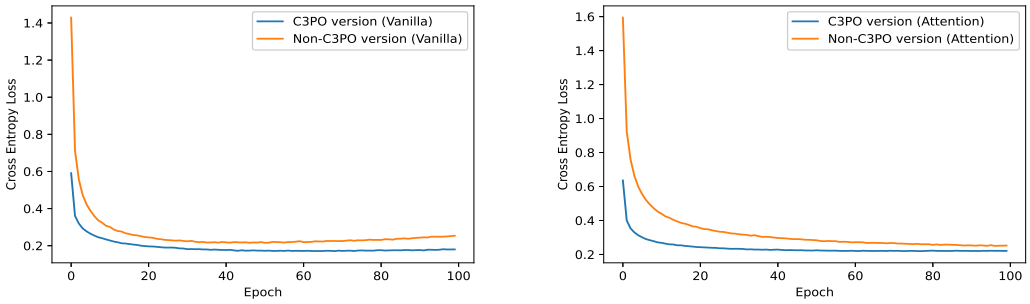

Figure 2: Average loss per epoch for the models and their 2 versions. (Left) Plot for Vanilla Seq2Seq versions, (Right) Plot for Attention Seq2Seq versions

- Token and its length
- If the token is numeric or alphabetical
- If the token is alphanumeric
- If the token is a punctuation mark
- The previous 2 tokens in that sentence
- The next 2 tokens in that sentence

The target or ground-truth prediction (0 or 1) for each token was generated by comparing the tokens of pseudocode sequence P and code sequence C, as a preprocessing step. If the current pseudocode token P_i also appeared in C, the target for binary tag sequence T_i would be 1. If it does not appear in C, the target would be 0. During inference, only the pseudocode sequence P is used and the features are passed as input to the decision tree.

5.3 Generate Model

To test the validity of the C3PO copy mechanism, it has been tested in conjunction with two different generate phase models – Vanilla Seq2Seq (Sutskever et al., 2014) and Attention Seq2Seq (Bahdanau et al., 2015). The models were built using PyTorch (Paszke et al., 2019). We will not review these popular architectures in detail.

For each generate model, two different versions have been trained. The first version, referred to as *non-C3PO version*, is trained on the non-masked input P with the target C. The second version, referred to as *C3PO version*, is trained with the masked tag sequence P_{masked} provided as input with target C_{masked}. The non-C3PO version provides a baseline to justify the C3PO version.

For the C3PO version, the input pseudocode vocabulary (denoted by $PVoc_{masked}$) is built after masking the copied tags, hence the copied tokens will not be included in the input vocabulary. This reduces the input vocabulary size to 30% of the original vocabulary size without masking (denoted by $PVoc$). Similarly, the output code vocabulary built after masking (denoted by $CVoc_{masked}$), also reduces the size to 20% of the original output vocabulary size without masking (denoted by $CVoc$) The vocabulary sizes are presented in Table 3

We additionally also experimented with *C3PO* and *non-C3PO* versions of a Vanilla Transformer model (Vaswani et al., 2017) trained from scratch.

5.4 Combine Model

To convert the C3PO model's output into the required code output, the masked tokens in the C_{masked} need to be unmasked. This is achieved by performing a one-to-one replacement of the masked tokens in their order of occurrence in P. If the model generates a `[CPY]` token, it is replaced with P_j which corresponds to one of the tokens in P that were masked ($P_j \in P_{cpy}$). If the model generated any other token, it is left as is.

$$Combined_i = \begin{cases} P_j & C_{masked_i} = \text{[CPY]} \\ C_{masked_i} & C_{masked_i} \text{ != [CPY]} \end{cases}$$

5.5 Experimental Setting

All our experiments were conducted using an NVIDIA GTX 1650 GPU with 4 GB of VRAM. The hyperparameters chosen for each Seq2Seq model for optimal performance on the GPU are as follows. For the Vanilla Seq2Seq models, an embedding size of 300, an LSTM hidden state size of 1024 and a batch size of 64 were chosen. For the Attention Seq2Seq models, an embedding size of 100, an LSTM hidden state size of 256 and a batch size of 32 were chosen. The hyperparameters were kept consistent across both versions (non-C3PO and C3PO) of each model. All models were trained

	Trainable Parameters		Training Time		BLEU	
Model (Version)	non-C3PO	C3PO	non-C3PO	C3PO	non-C3PO	C3PO
Vanilla Seq2Seq	20.3 M	**12.8 M**	21 h	**13 h**	0.44	**0.51**
Attention Seq2Seq	4.5 M	**2.5 M**	12 h	**7 h**	**0.75**	0.69
Vanilla Transformer	18.7 M	**11.7 M**	13 h	**10 h**	0.01	**0.19**

Table 2: Comparing non-C3PO and C3PO versions of the two models, in terms of the number of trainable parameters (in millions), training times (in hours) and BLEU score

for 100 epochs, using the Adam optimizer with a learning rate of 0.001, Cross-Entropy loss and teacher-forcing rate of 0.5.

For the Vanilla Transformer model, we used a batch size of 32, an embedding size of 512, 8 attention heads and 3 encoder and decoder layers each.

6 Results

BLEU-4 score (Papineni et al., 2002) is chosen as the evaluation metric and reported in Table 2, along with the model's training time and parameters. Some example translations generated by the C3PO model are demonstrated in Table 1.

The BLEU score for the non-C3PO version was 0.44 and for the C3PO was 0.51. Therefore we can say that using the C3PO mechanism, the model performed relatively better than the non-C3PO version. As for the Seq2Seq with Attention model, the BLEU score for the non-C3PO version is 0.75 and for the C3PO version is 0.69. In this case, the non-C3PO version performs better, but it comes at the cost of high training time and model size.

Further, we point out the significant difference in the number of parameters, and hence the training time, for both the versions in Table 2. The C3PO version has significantly fewer parameters owing to the decrease in vocabulary size. For the Vanilla Seq2Seq, this is a win on two counts; it achieved a higher BLEU score and it was more efficient.

The BLEU scores of the non-C3PO and C3PO Transformers are 0.01 and 0.19, respectively. Since transformers are data and compute-heavy architectures and we trained them on limited resources, they perform much worse than the RNN models.

In Fig 2, we can notice that in both Vanilla and Attention models, the C3PO version converges much faster than the non-C3PO version. This would lead us to believe that the C3PO versions would perform similarly, even if they were trained for lesser epochs than the non-C3PO version. It should also be noted that while the C3PO versions perform similarly, if not better, than the non-C3PO versions, they do so while using only 20% of the original vocabulary for both inputs and outputs, as shown in Table 3. This shows that the C3PO versions are both computationally efficient as well as data-efficient.

Version	Pseudocode	Code
non-C3PO version	6495	5647
C3PO version (ours)	**1984**	**1080**

Table 3: Input (pseudocode) and Output (code) vocabulary sizes in the two versions

7 Auxiliary Experiments

7.1 Numbered CPY tags

A possible problem with the C3PO is that the usage of the naive algorithm for combining in Section 5.4, assumes the order of copied tokens is the same in both pseudocode and code. To handle this, an attempt was made to use the Seq2Seq model itself to generate the CPY tags and put them in the right order. Instead of using a single `[CPY]` token, each unique variable was given a token `[CPY_n]`, where n is a unique number. For the example in Fig 1, the C_{masked} sequence would instead be `create integer [CPY1] [CPY2].`

With numbered tags, the Vanilla Seq2Seq and Attention Seq2Seq obtained BLEU scores of **0.54** and **0.66** respectively. As this is only a marginal difference from the initial results, numbering the tags doesn't offer a significant benefit. However, it is interesting to note that the BLEU score increases for Vanilla Seq2Seq but reduces for Attention Seq2Seq when tags are numbered.

7.2 Pretrained CodeT5 model

A pretrained CodeT5 model was used on the text data (without CPY tags). It was fine-tuned on our data from 3 hours (4 epochs) using beam search. The results were encouraging, with a BLEU score

of 0.85. The disadvantage of this approach lies only in its space and time complexity, which is suited for large datasets and high-performance compute. For computational efficient training on commodity hardware and a small amount of data, the C3PO versions of Attention and Vanilla Seq2Seq would be the best choice.

8 Conclusion and Future Work

We have introduced C3PO, a copying mechanism that emphasises computational and data efficiency. The methodology exploited the property of code where most tokens remain consistent across input and output. By masking such tokens, the vocabulary sizes are reduced significantly, which also reduced the training times. In future works, the method for filling masked tokens in output can be improved to fill the tokens while handling cases where the assumption that the order of masked tokens would remain consistent fails.

References

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 208–212.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

GitHub. 2021. Github copilot. https://github.com/features/copilot. Accessed: 2022-09-06.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code generation from natural language with less prior knowledge and more monolingual data. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 776–785, Online. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. 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. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. 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.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems*, 28.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.