Improving Code Quality through Fine-Tuning Code Generation Models using Reinforcement Learning

Johannes Hagemann
Hasso-Plattner-Insitute
University of Potsdam
johannes.hagemann@student.hpi.de

Abstract

Code generation models integrated into IDEs have recently gained popularity in the developer community. Most of the models lack a high quality in the produced code snippets that they output. In this work, we attempt to control the output of one of these code generation models so that it produces higher quality code snippets. For this purpose, we build a reward model that predicts the code quality of a code snippet for a given question, and use this model as a reward function to fine-tune a pre-trained code generation model using reinforcement learning. Our reward model achieves an 86% accuracy in distinguishing between the StaQC dataset of StackOverflow code snippets and our artificially created buggy/bad code snippets. Using this reward model, we attempt to steer the output of the code generation model to create higher quality code snippets.

1 Introduction

Recent breakthroughs in large language model development [5] have created interest in using the vast amount of publicly available code from sites like Stack Overflow or GitHub to fine-tune large language models on this data. This has led to code generation models such as Codex [6] or CodeParrot [3], which produce state-of-the-art results in code completion.
The potential productivity gains from using code generation models in a software developer's workflow is currently driving the integration of such models into IDE environments such through the popular VSCode plugin GitHub CoPilot, which is based on the code generation model Codex [6]. The increasing use of code generation models in programmer workflows highlights the importance of ensuring that the code generated by the models conforms to the standard design practices of the used programming language and does not generate buggy code.
This works explores a method to steer a code generation model to produce Python code with higher code quality and and less bugs. For this, we make use of a novel technique to fine-tune a code generation model called CodeParrot[1] using Reinforcement Learning.
The fine-tuning process works as follows. The model receives a coding question in natural language, similar to the kind one might find on a Stack Overflow post, and is tasked to produce a code snippet that solves this problem. To reward high quality and correct code snippets, we use a BERT classifier to analyze the quality of the produced code snippet, and use the classifier outputs as reward signals for the reinforcement learning training which is performed using the PPO algorithm. In summary, our main contributions are listed as follows:

1. We take a Stack Overflow question-code dataset and augment it with artificially created bad/buggy code. We publicly release this dataset for further research.
2. We achieve an accuracy of 86% in distinguishing between good and bad code snippets using our BERT classification model.

[1] *CodeParrot:* huggingface.co/codeparrot

3. We show that fine-tuning code generation models using reinforcement learning is a promising approach to steer those models into producing higher quality and less buggy code.

We provide the code and models for this work in a google drive folder. Please follow the provided link[2].

The paper is organized as follows. Section 2 presents the related work. Section 3 introduces the Stack Overflow code-question dataset called StaQC and explains how we extend this dataset to create artificially bad code snippets. Then, in section 4, the general approach, the training of the reward model, and the fine-tuning of the code generation model are presented. Building on this, section 5 shows the results of our experiments, while section 6 points out the limitations of this work and some promising future work directions. Finally, 7 provides a summary of our main findings.

2 Related Work

Recent work has focused on leveraging the vast amount of publicly available code on the internet to build deep-learning based code generation models [6, 10, 8, 17]. Many of the models are at the time of this writing not publicly accessible for fine-tuning, such as the code generation model Codex, which powers the popular VSCode extension GitHub CoPilot. However, several alternative open source code generation models have recently appeared, such as CodeParrot [3], which focuses on code generation in Python, or CodeGen [10], which is a model with a similar performance to Codex.

The other branch of recent work on which this paper builds up on is based on techniques for fine-tuning language models with reinforcement learning using a reward model [19]. Often these reward models are trained on human preferences for specific text continuations. For example, Ziegler et al. [19] trained a reward model on human preferences to score the quality of a summary and then fine-tuned GPT-2 with reinforcement learning using this reward model. Since then, the same approach has been used to fine-tune large language models on human preferences for summarizing entire books [16], training a language model to use a web browser environment to more accurately answer open-ended questions [9], or aligning language models to follow instructions[11].
This work builds up on the Transformer Reinforcement Learning (trl) library [4] which provides an implementation to train transformer language models with Proximal Policy Optimization (PPO) as it was proposed in the "Fine-Tuning Language Models from Human Preferences" paper by Ziegler et al. [19].

3 Dataset

The following section first introduces the Stack Overflow code-question dataset that we use in the training of the BERT classifier model as good training examples. Afterwards, we explain how we augmented this data to artificially create bad code snippets that we were able to use as negative training examples for the BERT classifier.

3.1 StaQC

The StaQC dataset consists of 148k Python and 120k SQL question-code pairs automatically mined from Stack Overflow.[18] Since the focus of this work is on code generation for Python, we only used this portion of the dataset. An example of a Python question in this dataset is the following:

How to grab from JSON in selenium

The matching Python code snippet for this question in the dataset is the following:

```
from bs4 import BeautifulSoup
import json

soup = BeautifulSoup(driver.page_source)
dict_from_json = json.loads(soup.find("body").text)
```

[2] drive.google.com/drive/folders/1iZevo7TgNXjukTh-AfuRxe$_vUZ8WnVzy$

In the following, we assume that these code snippets represent the "good" examples for the training of the reward model.

3.2 "Artificially" Buggy Code

Inspired by the work of Pradel et al. [12] who trained a bug detection classification model for JavaScript code by artificially generating buggy code through simple code transformations on an existing JavaScript code dataset, we applied a similar approach to artificially generate buggy and incorrect code through code transformations on the StaQC Python dataset.
Based on examples and principles from various online resources [1, 13] that explain bad practices of writing Python code, we introduced several Python anti-patterns, i.e. "non-pythonic" ways of writing code, into the samples to create code snippets which have a lower quality than the original ones. We focused on Python anti-patterns that are easy to implement via rule-based pattern matching. Among others, we introduced the following Python anti-patterns and randomly applied them to the code snippets from the StaQC dataset:

- We add non-explicit variable and function names. For example, by replacing all variables in the code snippet with one letter variable names. The following variable declaration,

  ```
  dict_from_json = json.loads(soup.find("body").text)
  ```

  would be converted into the following non-explicit format:

  ```
  y = json.loads(l.find("body").text)
  ```

- We turn variable and function names into camel case. For example, we turn the following function definition,

  ```
  def compute_net_value(price, tax):
  ```

  into the following format:

  ```
  def computeNetValue(price, tax):
  ```

- We remove all comments from the code snippet.

Additionally, for 25% of the original StaQC question-code pairs, we completley break the question-code relationship by combining questions with code snippets that do not match. For example, the following question:

How to rearrange Pandas column sequence?

will be combined with a code snippet answer that does not match the specified question at all:

```
result.append(pool.apply_async(worker, [job]))
```

Combining these different techniques, we obtain a dataset with 50k bad Python code snippets. In the next step, we use these code snippets to train the BERT classifier model.

4 Methods

The next section outlines our general approach to fine-tuning a code generation model to produce higher quality code, while the following two sections elaborate on the two models used in this approach.

4.1 General Approach

We are fine-tuning the 110M parameter code generation model CodeParrot to generate less buggy and higher quality code snippets with reinforcement learning. The optimization process can be described as follows. We prompt the CodeParrot model with a coding question in natural language to let it produce a code snippet that solves this question. Next, we insert the question and generated code snippet into a BERT classifier model that we have fine-tuned to predict the quality of

a question-code snippet pair, and use its output as a reward signal for the reinforcement learning training performed with the PPO algorithm.

The training loop, also shown with an example in Figure 1, can be summarized to the following three steps:

1. **Rollout:** We generate the query responses from the policy network, which is the code generation model that we are fine-tuning.
2. **Evaluation:** We obtain the quality scores as a scalar value between 0 (bad code snippet) and 1 (good code snippet) for the query and generated code snippet from our fine-tuned BERT model.
3. **Optimization:** We take the question and code snippet responses and input them into the CodeParrot model that we are optimizing as well as into a copy of the original CodeParrot model to determine the log-probabilities of all tokens in the sequence for both models. Then, we compute the KL-divergence between the two log-probability outputs of the models, which will be used as an additional reward signal to ensure that the optimized model does not diverge too far from the original code generation model. Finally, we optimize the policy with PPO using the $(query, response, reward)$ triplet.

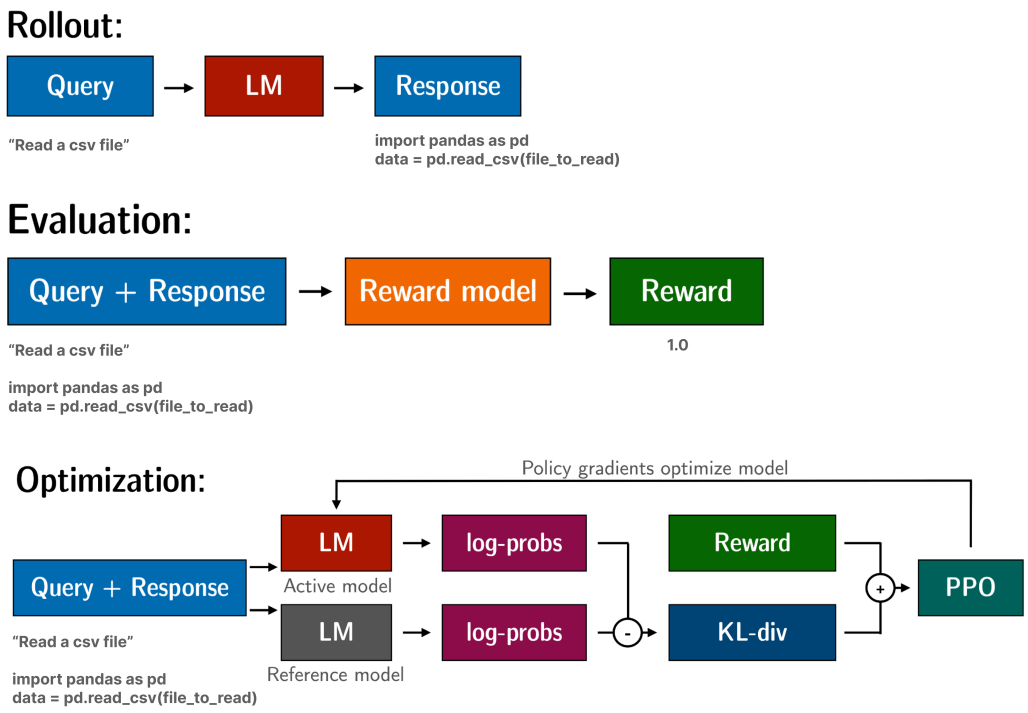


Figure 1: Sketch of the worfklow by [4] adjusted for our code generation scenario

Next, the training process of the BERT classifier model is explained in detail.

4.2 Reward Model

For the reward model, we fine-tune a distilled version of BERT [7] called DistilBERT [15] on the good and bad code snippets introduced in Section 3 to predict a score for the quality and correctness of a question-code snippet pair. We chose DistilBERT because it is a smaller and faster version of the BERT model, which resulted in shorter training times. The good question-code-snippet pairs from the StaQC dataset are given a label of 1 and the artificially generated low quality and incorrect code snippets are given a label of 0. The combined size of the dataset of good and bad code snippets

resulted in about 135k samples, which we split up into a 90:5:5 train-validation-test split. We fine-tune the model with a batch size of 16 and a learning rate of 2×10^{-5} for a total of 4 epochs.

4.3 Transformer Reinforcement Learning

For the Transformer Reinforcement Learning part, we fine-tuned the CodeParrot model with 110M parameters according to the steps introduced in Section 4.1. After experimenting with larger open-source Python code generation models such as the CodeParrot model with 1.5B parameters [2], we switched to the 110M parameter model due to compute constraints. The implications of this decision are discussed in the Limitations section. The 110M parameter CodeParrot model follows the GPT-2 architecture [14], has a context size of 1024 tokens, and has been trained on a total of 29 billion tokens of mainly Python code[3].

An important aspect of this fine-tuning process, already briefly mentioned for the optimization step, is that we do not want the outputs of the model to deviate too far from the original, pre-trained code generation model. For this purpose, we calculate the KL divergence between the two log-probability outputs of the policies for the optimized model π and the original model ρ and add it as a penalty to the reward r from the reward model. Thus, the modified reward that we use in the PPO optimization step is the following [19]:

$$R(x, y) = r(x, y) - \beta \log \frac{\pi(y|x)}{\rho(y|x)} = r(x, y) - \beta KL(\pi, \rho). \tag{1}$$

β can either be selected as a constant or dynamically adapted.
This process ensures that the text and code the model creates still makes sense and is not just optimizing to get a high reward from the reward model.

Using the TRL library and its PPO trainer component [4], we fine-tune the code generation model using reinforcement learning for 175 iterations with a batch size of 64 and a learning rate of 5×10^{-6}. During this fine-tuning process, we first generate code snippets for each question in our batch using the code generation model that we are optimizing. We limit the length of those code snippet generations to 128 tokens. Then, we produce the rewards for each question and generated code snippet in the batch using the reward model. Finally, we call the step function of the PPO trainer component which optimizes the code generation model using the $(query, response, reward)$ triplet.

5 Results

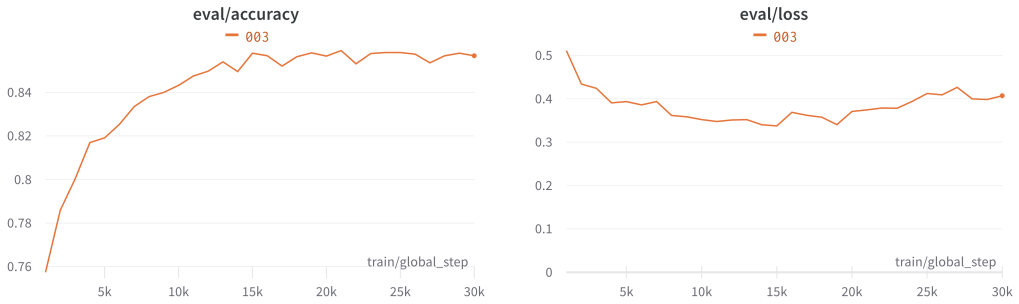
In the following, we report the results of our experiments. First for the fine-tuned reward model and then for the fine-tuned code generation model using the reward model.

Reward Model After training the model for 30k steps to distinguish between good and bad code snippets, we achieve a validation accuracy of about 86%. However, the validation loss curve of the training run suggests that we already overfitted on the training data after 15k step or 2 epochs (see Figure 2).

A qualitative analysis shows that the model is able to detect the rules and design patterns that we have defined as bad coding practices, such as camel case or non-explicit function and variable names. For example, the following question-code snippet pair, which contains a variable in camel-case format, is predicted to be a bad code snippet with a probability of 98.8%:

```
# Read a csv file
import pandas as pd
data = pd.read_csv(fileToRead)
```

However, the model does not generalize to other bad coding practices or mistakes that were not covered by our data augmentation. For example, the following code snippet, which many programmers would easily recognize as a bad code snippet for the given question since it directly



(a) Accuracy on validation dataset

(b) Validation loss curve

Figure 2: Evaluation metrics for fine-tuning the reward model.

drops all the data after loading it, is classified as a good code snippet with a probability of 99.8%:

```
# Read a csv file
import pandas as pd
data = pd.read_csv(file_to_read)
data.drop(data.index, inplace=True)
```

Transformer Reinforcement Learning After fine-tuning the model for 175 steps, we could not see any improvements of the model to optimize for the reward model. The mean reward of the outputs seem to stagnate at around a score of 0.6 or even seem to get worse as Figure 3 shows. A qualitative analysis also shows that the outputs of the optimized model seem to get more repetitive instead of actually optimizing for better code quality and less bugs.

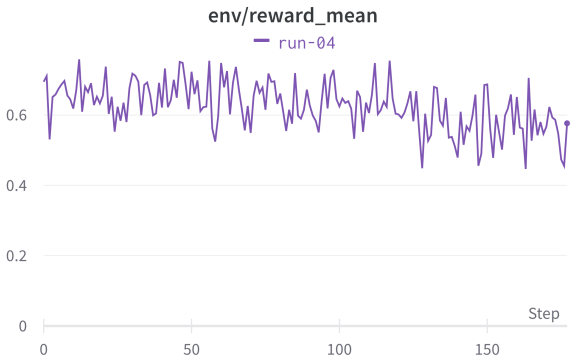
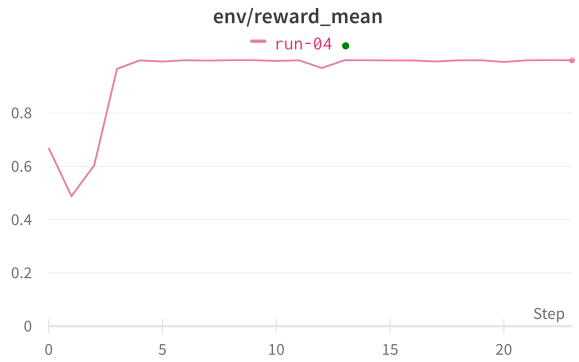


Figure 3: Mean reward of the optimized code generation model during training for a question and generated code snippet

We also experimented with different learning rates and thresholds for the allowed KL divergence between the model we optimized and the reference code generation model. Unfortunately, when we set these values too high, the model diverged too much from the reference code generation model and outputs the same string for each question, optimized only for a high reward of the reward model. Figure 4 shows such a run with a higher learning rate of $5 \times 10{-}5$. Although the mean reward converges to 1 quite quickly, so that according to the reward model the code generation model is perfectly optimized to produce only good code snippets, all generated code snippets are the same random string.

In general, the initial positive results of the reward model for distinguishing between good and bad code snippets could not yet be translated into an actual optimization of the code generation model through fine-tuning it with reinforcement learning.



(a) Mean reward of a question and generated code snippet

	query	generated_code_snippet	reward
1	# Passing Partial Lists into Python Function	# Passing Partial Lists into Python Function NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9974
2	# Python min override	# Python min override NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9976
3	# Generate a 2D boolean array from tuples	# Generate a 2D boolean array from tuples NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9987
4	# from pyqt4 designer gui how to get radio button value using python	# from pyqt4 designer gui how to get radio button value using python NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9985
5	# Only print max value (python)	# Only print max value (python) NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9984
6	# python/beautifulsoup to find all <a href> with specific anchor text	# python/beautifulsoup to find all <a href> with specific anchor text NSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNSNS	0.9985

(b) Example generated code snippets and its rewards.

Figure 4: Transformer Reinforcement Learning training run with a too high learning rate.

6 Discussion

6.1 Limitations

Although our reward model achieves a high accuracy in discriminating between good and bad code snippets, we found that it has a number of limitations. Also, our attempt to optimize a code generation model using this reward model has not proven effective, and we will mention some possible reasons for its poor performance in this section.
First, the code snippets in the StaQC dataset that we used as ground truth good labels for fine-tuning the reward model are of rather poor quality. Since the dataset consists only of scrapped responses from StackOverflow, many of the code snippets contain unnecessary examples of the posted question.
Our data augmentation to artificially generate bad code snippets is not good enough. As a result, we were only able to teach the model some simple rules that are classified as bad coding practices, but we did not see any generalization to identify other bad coding practices or buggy code.
Both concerns put a limit on how good of a proxy the reward model actually is in classifying the quality of a code snippet. In our case, this proxy, which mainly represents a few bad coding practices, is certainly not good enough. Creating a new dataset for good and bad code snippets, as described in the Future Work section, would address this issue.

Another limitation for fine-tuning the code generation model came from the small model size we used due to compute constraints. The code generation model with 110M parameters does not have a really good performance out of the box, which makes it hard to decide if one can improve the results of the model in any meaningful way.

6.2 Future Work

The deficiencies of our current results could be improved in future work through improvements to the underlying code quality classification model used as the reward model. These improvements would likely be achieved primarily through improvements to the dataset for good and bad code snippets.
One approach for generating this data could be to use a better performing code generation model such as Codex [6] and prompt it via few-shot examples to produce good and bad code snippets for a set of coding questions. Additional labeling through human feedback would be necessary. Based on this approach, we would hopefully obtain higher quality good code snippets and more diverse bad code snippets.

Another future direction of work could be to simply try the approach on a larger code generation model with better performance, such as the CodeParrot model with 1.5 B parameters.

An interesting ablation for future work on the project would be to see how the Transformer Reinforcement Learning approach performs against traditional fine-tuning using a supervised dataset.

7 Conclusion

We investigated whether it is possible to fine-tune a code generation model using reinforcement learning and a reward model to produce higher quality code.
Our results are mixed. The reward model properly learned to distinguish between the data augmentations we've defined as good and bad code snippets. However, fine-tuning the code generation model using the reward model only led to the optimized code generation model outputting redundant code snippets. The performance of the model could be improved if more time and resources were invested in a broader representative dataset of good and bad code snippets.
Nevertheless, we view the presented research direction as a promising approach to steer code generation models in a way that produces higher quality and less buggy code.

References

[1] 18 common python anti-patterns i wish i had known before. https://towardsdatascience.com/18-common-python-anti-patterns. Accessed: 2022-08-29.

[2] Codeparrot 1.5b parameters. https://huggingface.co/codeparrot/codeparrot. Accessed: 2022-08-30.

[3] Training codeparrot from scratch. https://huggingface.co/blog/codeparrot. Accessed: 2022-08-28.

[4] Transformer reinforcement learning library. https://github.com/lvwerra/trl. Accessed: 2022-08-28.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[9] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2021.

[10] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis, 2022.

[11] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[12] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[13] QuantifiedCode. *Python Anti-Patterns*. 2018.

[14] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[15] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019.

[16] Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback, 2020.

[17] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. *CoRR*, abs/1803.09371, 2018.

[19] Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences, 2019.