InGn: A Code Generative Model

Arif Muhammed
Department of Computer Science and Engineering
Amal Jyothi College Of Engineering
Kanjirappally, Kotayam, Kerala, India
arifmuhammed2023@cs.ajce.in

Asif Shereef
Department of Computer Science and Engineering
Amal Jyothi College Of Engineering
Kanjirappally, Kotayam, Kerala, India
asifshereef2023@cs.ajce.in

Ben Jacob Bobby
Department of Computer Science and Engineering
Amal Jyothi College Of Engineering
Kanjirappally, Kotayam, Kerala, India
benjacobbobby2023@cs.ajce.in

Chetan Manoj
Department of Computer Science and Engineering
Amal Jyothi College Of Engineering
Kanjirappally, Kotayam, Kerala, India
chetanmanoj2023@cs.ajce.in

Dr.Sinciya P.O
Department of Computer Science and Engineering
Amal Jyothi College Of Engineering
Kanjirappally, Kotayam, Kerala, India
posinciya@amaljyothi.ac.in

***Abstract*—The INGN is an advanced generative model that can generate and modify code. It has been trained on a vast database of permissively licensed code, in which certain sections of the code have been masked and relocated to the end of the file. This distinctive feature enables the model to perform "infilling," which refers to the process of code completion using bidirectional context. In contrast, INCODER is the first major generative code model that can infill arbitrary regions of code, and it has been evaluated on various tasks, such as type inference, comment generation, and variable renaming in a zero-shot setting. The INCODER model has demonstrated exceptional performance in these evaluations, particularly when compared to left-to-right only models that have undergone similar pre-training. Furthermore, the INCODER model and code are available for public use.**

***Index Terms*—Code Generation, Doc String, Infilling.**

I. Introduction

Large language models trained on large amounts of code have made significant progress in neural program synthesis tasks, such as automatic code generation. However, these models typically generate code from left to right, which makes them less effective in completing common code editing tasks such as bug fixing, adding comments, or variable renaming. To address this issue, INGN, a unified model that can generate and edit code, has been developed by researchers. Like other models, INGN is trained to maximize the likelihood of a corpus of code but uses a causal masking objective to enable it to infill blocks of code using both left and right context, making it more versatile and applicable to a wider range of code editing tasks.

To train INGN to perform code infilling, researchers replace random spans of code with a special placeholder token and move them to the end of the code sequence (as shown in Figure 1, top). The model is then trained to predict all the tokens in the modified sequence. During the infilling process, the model can be prompted with a new sequence that includes placeholder tokens in the sections of code that need to be edited or completed. The model generates new tokens to replace the placeholder tokens (as shown in Figure 1, bottom), enabling INGN to both generate code from left to right (program synthesis) and edit code by infilling. It is a unified approach that can be used for both tasks.

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        <MASK:0> in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
        for line in f:
            for word in line.split():
                if word <EOM>
```

Fig. 1. Original Document and Masked Document

Researchers evaluated INGN's performance on various zero-shot code infilling tasks, including both new and previously studied tasks. These tasks include predicting types, renaming variables, generating comments, and completing missing lines of code. The researchers found that using INGN to perform infilling with bidirectional context significantly outperformed approaches that use models that can only generate code from

left to right. On several tasks, INGN performed similarly to state-of-the-art models fine-tuned specifically for those tasks. Additionally, INGN performed well on existing program synthesis tasks that require code to be generated from left to right, despite being trained on a more general objective. Overall, INGN demonstrated strong performance on a range of code infilling tasks.

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_count = {}
        for line in f:
            for word in line.split():
                if word in word_count:
                    word_count[word] += 1
                else:
                    word_count[word] = 1
    return word_count
```

Fig. 2. Type Inference and Variable Name Prediction

Expanding on this topic, INGN represents a significant advancement in the field of generative code models. By enabling both code generation and editing, INGN is an innovative solution to the limitations of previous models that could only generate code from left to right. INGN's ability to perform code infilling using bidirectional context is particularly noteworthy, as it allows the model to complete common code editing tasks that were previously challenging for other models.

The approach used to train INGN is also significant, as it allows the model to learn to infill code by predicting all the tokens in a modified code sequence. By using a causal masking objective, the model can effectively use both left and right context to perform infilling, making it more versatile and adaptable to a wide range of code editing tasks.

The approach used to train INGN is also significant, as it allows the model to learn to infill code by predicting all the tokens in a modified code sequence. By using a causal masking objective, the model can effectively use both left and right context to perform infilling, making it more versatile and adaptable to a wide range of code editing tasks.

II. Literature Survey

Program synthesis is an exciting area of computer science that aims to automate the process of creating computer programs, potentially revolutionizing the way we think about programming. In traditional programming, humans manually enter code, but program synthesis automates this process by generating a computer program that meets the user's desired specifications. This has the potential to improve the productivity of experienced programmers and make programming more accessible to a wider audience.

```
def count_words(filename: str) -> Dict[str, int]:
    """
    Counts the number of occurrences of each word in the given file.

    :param filename: The name of the file to count.
    :return: A dictionary mapping words to the number of occurrences.
    """
    with open(filename, 'r') as f:
            word_counts = {}
            for line in f:
                for word in line.split():
                    if word in word_counts:
                        word_counts[word] += 1
                    else:
                        word_counts[word] = 1
        return word_counts
from collections import Counter

def word_count(file_name):
    """Count the number of occurrences of each word in the file."""
    words = []
    with open(file_name) as file:
        for line in file:
            words.append(line.strip())
    return Counter(words)
```

Fig. 3. Docstring Generation and Multi-Region Infilling

However, successful program synthesis is challenging due to the vastness of the search space and the difficulty of accurately specifying user intent. Researchers have proposed a multi-turn approach in which the user and the synthesis system work together to complete a program in multiple steps. This approach allows the user to break down a potentially long and complex specification into smaller, more manageable steps, making it easier for the model to understand and improve the quality of the synthesized program.

To evaluate the performance of models on multi-turn program synthesis tasks, researchers have developed a benchmark that requires a model to synthesize a program in multiple steps, with the user specifying their intent in each step in natural language.

INCODER is a unified model that performs both program synthesis and editing, addressing the limitations of left-to-right only models. It infills arbitrary code regions and considers bidirectional context, improving performance on various zero-shot code infilling tasks. The ability to consider bidirectional context has the potential to improve performance on other program synthesis tasks as well.

As large language models continue to develop, the potential of program synthesis will only continue to grow. Successful program synthesis has the potential to make programming more accessible and improve the productivity of experienced programmers, making it a highly desirable goal for computer scientists.

Moreover, program synthesis has numerous applications across various domains, including robotics, software engineering, and natural language processing. For instance, in robotics, program synthesis can be used to generate complex control policies for robots that accomplish a specific task. In software engineering, program synthesis can help developers create more efficient code and reduce the time and effort required

for testing and debugging. In natural language processing, program synthesis can aid in the development of chatbots and virtual assistants that can understand and respond to natural language queries.

The potential impact of program synthesis is not limited to industry applications. It can also have a significant impact on education and research. For example, program synthesis can make it easier for students to learn programming by providing them with a tool that can automatically generate code from natural language descriptions. Additionally, program synthesis can help researchers in developing new programming languages and tools by automating the process of generating code in these languages.

However, there are also ethical considerations that need to be taken into account when it comes to program synthesis. One major concern is the possibility of automated code generation leading to job displacement for programmers. While program synthesis has the potential to increase productivity, it may also lead to the loss of jobs for programmers who are currently employed to write code manually. Additionally, there are concerns about the ethical implications of using program synthesis to generate malicious code or code that violates privacy and security.

In conclusion, program synthesis is a promising field that has the potential to revolutionize programming and improve productivity. While there are challenges that need to be addressed, such as accurately specifying user intent and ensuring ethical implications are taken into account, recent advancements such as the multi-turn approach and models like INCODER show promise in overcoming these challenges. With further research and development, program synthesis has the potential to become a game-changer for the software industry, education, and research.

III. Proposed System

Until recently, generating code has been a challenge for AI models due to the complex nature of programming languages and the various contexts in which they can be used. Two popular approaches for generating code are left-to-right language modeling and masked language modeling, but each has its own limitations. Left-to-right modeling generates complete documents, but it is not suitable for infilling, while masked language modeling is ideal for infilling, but it can only generate a small portion of a document (usually around 15%).

To overcome these limitations, researchers have developed a new approach called causal masking, which combines the benefits of both methods. This approach allows the model to consider both the left and right context when performing infilling, while still being able to generate complete documents. In other words, the model can generate new code by filling in missing pieces in a program or by suggesting modifications to an existing program, while still considering the context of the entire codebase.

The causal masking approach works by creating a mask that only allows the model to access information that precedes the current token during training, but not information that follows it. This enables the model to generate code in a way that is consistent with the natural flow of programming languages, while also considering context from both directions.

By adopting the causal masking approach, researchers have been able to develop models that can generate high-quality code while also supporting infilling. These models have shown significant improvements in generating code for a variety of tasks, such as fixing bugs, renaming variables, and adding comments. Moreover, these models can be trained on large datasets of code, enabling them to learn patterns and idioms commonly used in programming languages.

In conclusion, the causal masking approach offers a promising solution to the challenges of generating high-quality code while also supporting infilling. This method allows models to consider both the left and right context when generating code, resulting in more natural and accurate code generation. As researchers continue to develop and refine these models, they will become increasingly useful for a variety of programming tasks.

A. Training

Programmers have long sought to automate the process of code generation. While left-to-right language modeling has been useful in generating entire documents, it does not support infilling, while masked language modeling allows infilling, but it is only capable of generating small portions of a document, around 15%. Researchers in the field have recently adopted a new method called causal masking, which combines the benefits of both approaches, allowing for complete document generation while still enabling the model to consider both left and right context for infilling.

During training, the causal masking process selects a certain number of spans of contiguous tokens from each document to mask. The number of spans is determined by a Poisson distribution with a mean of one, but the distribution is truncated to the range of 1 to 256, which means that there are generally a small number of spans with a single span about 50% of the time, but the distribution has a long tail that can go up to 256 spans. To prevent any overlap between spans, the selected spans are rejected and re-selected. Once the spans have been chosen, each span k is replaced with a special mask placeholder token, Mask:k.

To perform infilling, the sequence of tokens within the selected span is moved to the end of the document, with the mask token inserted in front of the moved span and a special end-of-mask token <EOM> appended at the end. In other words, when the mask token appears for the first time in left-to-right order, it indicates the location where the span was removed from; when it appears for the second time, it marks the start of the moved span text. Given a document D with N tokens and a selected span Span Dij, the left context is

represented by Di and the right context by Dj+1. The masked document M is given by the expression M = Left Mask:1 <EOM>. Right Span <EOM>.

This method enables the model to consider both left and right context when performing infilling, which is a significant improvement over previous approaches that only considered left or right context. The causal masking approach also allows the model to generate complete documents, making it more useful than the masked language modeling method, which can only generate small portions of a document.

$$\log P(\texttt{[Left; <Mask:0>; Right; <Mask:0>; Span; <EOM>]})$$

Fig. 4. Formula for Model Training

To further elaborate on the creation of the masked document M, it is important to note that after the selection of spans and replacement with the mask placeholder token, the left and right contexts are concatenated to form the masked document. This concatenated sequence consists of the tokens preceding the selected span, followed by the mask placeholder token, and then the tokens succeeding the selected span. For example, if the selected span in a document D is from tokens Di to Dj, the masked document M would be created by concatenating the left context Di, the mask placeholder token Mask:k, and the right context Dj+1.

In cases where more than one span is selected, they are appended at the end of the document in the order in which they were selected. This is done to ensure that the model has access to all the necessary context when generating the tokens to fill in the masked portions of the document.

Once the masked document M is created, the model generates the missing tokens in an auto-regressive manner. In other words, it generates the missing tokens one at a time, taking into account the tokens that have been generated so far. The probability of each generated token is calculated using a left-to-right language model, similar to traditional language generation models.

To train the model, cross-entropy loss is used on all tokens except for the mask placeholder tokens. This is done to ensure that the model learns to generate the correct tokens in the masked portions of the document while still maintaining coherence with the surrounding text. By excluding the mask placeholder tokens from the loss calculation, the model is incentivized to generate the correct tokens in their place during the infilling process.

B. Inference

In the infilling process, the code generating model has the flexibility to either generate code in a conventional left-to-right manner, or to insert code at any position in an existing document by using the Mask:k token at the desired location(s) and continue the generation process towards the end of the document. If we want to insert text at a single location, we can generate a span to be inserted by sampling tokens from the model's distribution in an auto-regressive manner, in between the left and right context sequences at the desired location.

By using the Mask:k token, the model knows where to insert the generated span, and can easily distinguish the inserted code from the original code. The insertion location can be chosen arbitrarily, providing the model with more flexibility to handle various programming tasks. During the generation process, the model calculates the probability of each generated token, and samples from the probability distribution to generate the next token in the sequence. The model is trained using cross-entropy loss on all tokens except the Mask:k tokens, ensuring that it doesn't generate these tokens during the infilling process.

$$P(\cdot \mid \texttt{[Left; <Mask:0>; Right; <Mask:0>]})$$

Fig. 5. Formula For Inference

The ability to insert code at any location in an existing document is a powerful tool for code generation. By inserting code in the middle of an existing document, we can make our generated code more context-aware and tailored to specific use cases. To insert text at a single location in a document, we first create a Mask:k token at the desired location. This signals the model to start generating text at that location. We can then sample tokens from the model's distribution in an auto-regressive manner to generate the desired code.

To insert text at multiple locations in a document, we create a mask sentinel token and append it to the end of the document. We then repeat this process for each location we want to insert code into. For example, if we want to insert code at two locations, we would create two mask sentinel tokens and append them to the end of the document. We can then fill in each mask with the appropriate text, and once a mask has been filled, we append the appropriate Mask:k sentinel token to signal the start of generation for the next span. The completed document would contain all of the inserted code, as well as the original code and mask sentinel tokens.

This technique allows us to use bidirectional context in code generation without the need for pre-training. We can leverage the context of existing code to generate Python docstrings based on function signatures and implementations. We can also generate import statements that are required by a function. By inserting code at multiple locations in a document, we can generate code that is more context-aware and tailored to specific use cases, making our code generation models more powerful and versatile.

IV. System Training

A. Training Data

The models presented in this paper were trained on a diverse data-set that includes code from various public repositories

with permissive, non-copyleft open-source licenses, as well as Stack Overflow content, such as questions, answers, and comments. The primary emphasis of this research is on the Python programming language, which constitutes 52 GB of the total corpus of 159 GB of code. However, the data-set also includes code files in 28 different languages, such as Java, C++, JavaScript, Ruby, and Go, to name a few. The Stack Overflow content accounts for a total of 57 GB of data, including questions, answers, and comments related to programming in various languages. To ensure data quality, the data-set was filtered and de-duplicated, resulting in a final set of high-quality code and content. A detailed breakdown of the data-set size by language is presented below:

TABLE I
CODE DATA BY LANGUAGE

Language	Code (GB)
Python	52
Java	17
C++	13
C#	8
JavaScript	6
Ruby	4
Go	2
Others	57

B. Code

1) Sources:

- Code files and repository metadata were obtained from GitHub and GitLab through the sites' public APIs. This data was collected until December 9th, 2021.
- Approximately 670,000 public non-fork repositories were obtained that contained primarily Python, JavaScript, or Jupyter Notebook files, and had either an MIT, Apache 2.0, BSD-2, or BSD-3 clause license.
- Code from a list of 28 languages was included in these repositories, based on file extension.
- Python files that were not already obtained from GitHub were also included from the GitHub archive on BigQuery.
- Jupyter notebooks were preprocessed by including all text and code, with Markdown formatting removed from text cells. Cells were demarcated by XML-style tags.

2) Deduplication:

- Recent research has indicated that model performance can be improved and the risk of memorizing training data can be reduced by removing duplicate training data
- The method we use for deduplicating training data is by removing code files that have an exact match on the sequence of alphanumeric tokens in the file.
- This method removed approximately 75% of the training data, reducing it from 1 TB to 250 GB. This was achieved by removing duplicated repositories, library dependencies, and common boilerplate code files, such as those used in Python web frameworks.
- To reduce the risk of the model memorizing real email addresses or hallucinating fake ones, we used regular expressions to detect email addresses in the code files and replaced them with dummy addresses.

3) Decontamination:

- To evaluate our code generation models on multiple current benchmarks, we perform data decontamination.
- The process involves removing any overlap between our training data and the evaluation sets of these benchmarks.
- Specifically, we remove any repositories that are present in the validation and test sets of CodeSearchNet.
- The reason for this is that these repositories are used to construct validation and test sets for several tasks in CodeXGLUE.

4) Filtering:

- Our filtering method for generative models of code is similar to previous research.
- We remove files that contain lines longer than 3000 tokens or have an average line length greater than 100 tokens.
- Additionally, we remove files that have less than 40
- We also exclude files that seem to be automatically generated.
- We identify automatically generated files by matching substrings with a small number of phrases.

C. Stack Overflow

The corpus used in the study is composed of two parts, with the second part consisting of questions, answers, and comments sourced from StackOverflow. This corpus serves as a benchmark for comparing recent generative code models, such as those trained on The Pile [26]. However, The Pile does not include comments, which are included in this corpus. The study extracts questions with at least one answer and selects up to ten answers per question based on their non-negative score, sorted in descending order of their score. Up to five comments for each question or answer are also included in the corpus. The model utilizes comments and the infilling ability to perform interactive code editing, guided by the natural language present in the comments. The model produces accurate and precise results.

D. MetaData

Metadata is incorporated in the training data for code files and StackOverflow questions/answers to enable attribute-conditioned generation and attribute prediction. The attributes for code file data include the code filename, file extension (used as a language proxy), file source (either GitHub or GitLab), and the number of stars for GitHub repositories, which are binned into six buckets. The metadata is optional during left-to-right prompting of the model and can be inserted at the beginning of the document with a 50% probability or at

the end of the document for metadata prediction. StackOverflow metadata attributes consist of the question tags for the topic (e.g., python, django) and the number of votes for each question and answer, which are also binned in the same way as repository stars. Comments are positioned directly after the questions or answers they were written for.

E. Tokenization

To increase the amount of context that our code model can condition on, the length of documents that the model can generate, and the efficiency of training and inference, we train a byte-level BPE tokenizer [56, 51]. We allow tokens to extend across whitespace (excluding newline characters) so that common code idioms (e.g., import numpy as np) are represented as single tokens in the vocabulary. This substantially improves the tokenizer's efficiency—reducing the total number of tokens required to encode our training corpus by 45% relative to the byte-level BPE tokenizer and vocabulary of GPT-2.

V. System

The INCODER-6.7B is a Transformer-based language model with a capacity of 6.7B that was used in a study. Its architecture is based on the dense 6.7B models that were previously described in a publication. During training, each training document was used once during one epoch, and the training process lasted for 24 days on 248 V100 GPUs. The training process used the causal masking approach provided in Fairseq and PyTorch as the underlying library. The per-GPU batch size was set to 8, and the maximum token sequence length was 2048. Gradient norms were clipped to 1.0, and the Adam optimizer with learning rate schedules was used. A built-in polynomial decay learning rate scheduler with 1500 warmup updates was used to schedule the learning rate. To improve memory efficiency through fully sharding model states, Fairscale was also used.

VI. Conclusion

Recent studies have demonstrated that including a causal masking objective during training of a generative code model can lead to impressive performance on challenging code infilling and editing tasks, even without explicit training on these tasks. However, these studies have also shown that the model's standard left-to-right generation ability is not compromised, with causal-masked models performing similarly to conventional models on traditional left-to-right language-to-code synthesis benchmarks. With further enhancements in model architecture, training data, and training time, it is expected that performance will continue to improve.

In addition to the improved infilling capabilities, the potential for fine-tuning the model offers promising prospects for improving its ability to interpret natural language instructions and other human intent indicators. By fine-tuning, the model could better understand the context in which a code snippet is required, resulting in more accurate and relevant code infilling and editing. Moreover, this model presents an opportunity for future research on supervised infilling and editing through model fine-tuning, which could result in even more remarkable zero-shot performance on a variety of code-related tasks.

The model's ability to perform iterative decoding is also a promising area for further study. By refining its own output, the model could adapt and improve based on the context in which it is used, leading to even more precise and accurate code generation. Overall, incorporating a causal masking objective during the training of generative code models presents a significant potential for advancing the field of code generation and related tasks.

References

[1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In Proceedings of EMNLP, 2019.

[2] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. CM3: A causal masked multimodal model of the Internet. arXiv preprint arXiv:2201.07520, 2022.

[3] Armen Aghajanyan, Dmytro Okhonko, Mike Lewis, Mandar Joshi, Hu Xu, Gargi Ghosh, and Luke Zettlemoyer. HTLM: Hyper-text pre-training and prompting of language models. In ICLR, 2022.

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pretraining for program understanding and generation. In NAACL, 2021.

[5] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In SPLASH, 2019.

[6] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In PLDI, 2020.

[7] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In ICML, pages 245–256. PMLR, 2020.

[8] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. arXiv preprint arXiv:2112.10684, 2021.

[9] Yannis M. Assael, Thea Sommerschield, and J. Prag. Restoring ancient text using deep learning: a case study on Greek epigraphy. In EMNLP, 2019.

[10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.

[11] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. FairScale: A general purpose modular PyTorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

[12] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In ICLR, 2017.

[13] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: neural-backed generators for program synthesis. PACMPL, 2019.

[14] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPTNeoX-20B: An open-source autoregressive language model. 2022.

[15] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970.

[16] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In NeurIPS, 2020.

[17] William Chan, Nikita Kitaev, Kelvin Guu, Mitchell Stern, and Jakob Uszkoreit. KERMIT: Generative insertion-based modeling for sequences. arXiv preprint arXiv:1906.01604, 2019.

[18] 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. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[19] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. SpreadsheetCoder: Formula prediction from semi-structured context. In ICML, 2021.

[20] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Monperrus Martin. SequenceR: Sequence-to-sequence learning for end-to end program repair. IEEE Transactions on Software Engineering, 2021.