Incoder: A Generative Model for Code Infilling and Synthesis

Presented by: Levent Toksoz

Motivation

Large Language Models that are trained on large code repositories shows great potential in neural program synthesis and other code related tasks

However most of these models are designed to generate code left to right

Thus they are not very effective at code editing tasks like fixing bugs, adding comments or re-naming variables

The ones that utilize Masked Language Modeling are mostly designed to infill very short spans

Motivation

Also this type of generation (left to right) is not natural for a human - typically we do not write a code in one left to right pass

Having ability to go back and edit some parts of the code is important

Hence they introduce Incoder - a model that is capable of infilling arbitrary regions of code

Incoder

Aim of the Incoder is to combine the strengths of both MLM and casual model

Want to take MLM`s ability to condition both left and right context (which is especially helpful for code infilling) and

Casual Model`s ability to autoregressively generate entire documents

Thus they introduce casual masking procedure

Casual Masking

At Training

Sample number of spans of contiguous tokens in each document from a Poisson distribution

Sample the length of each span uniformly from the length of the document (make sure spans do not overlap)

Replace each span k with special mask sentinel token (<Mask:k>)

Casual Masking

Then move the sequence of tokens that are in the span to the end of the document and mark the beginning of this span with the mask sentinel token and end with the end of mask token (<EOM>)

By doing this we want to maximize the log probability of the masked document:

$$\log P([\texttt{Left};\ \texttt{<Mask:0>};\ \texttt{Right};\ \texttt{<Mask:0>};\ \texttt{Span};\ \texttt{<EOM>}]) \tag{1}$$

Casual Masking

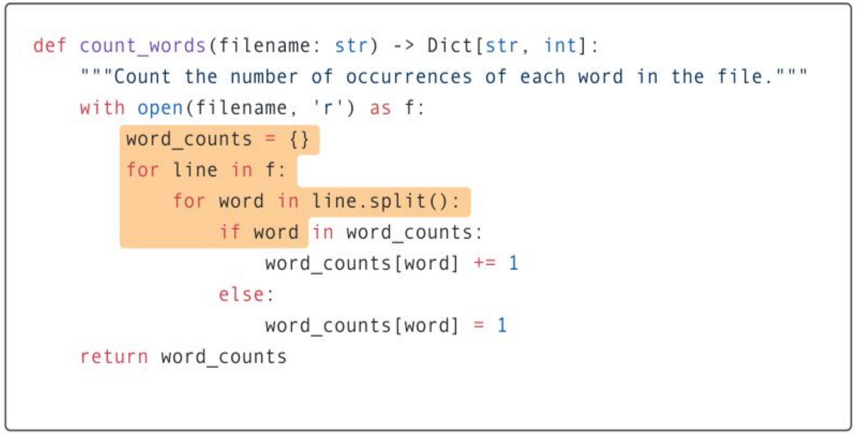
Thus computing the probability of the sequence auto-regressively

Model is trained on cross entropy loss on all tokens except the mask sentinel tokens (<Mask: k>) since we do not want the model to generate these tokens during the inference

Casual Masking

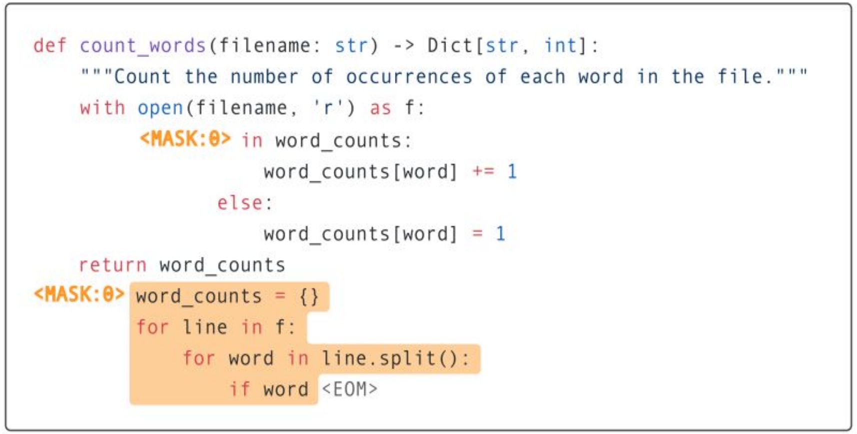
Training

Original Document



```
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
```

Masked Document



```
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>
```

Casual Masking

At Inference

Model can either used to generate left to right by sampling autoregressively

Or to insert code at any desired location by placing <Mask:k> token in that location and doing generation at the end of the document to get bidirectional context

$$P(\cdot \mid [\texttt{Left; <Mask:0>; Right; <Mask:0>}]) \tag{2}$$

Training

Data: open source public libraries from Github and Gitlab (mainly python), StackOverflow questions/answers/comments and code

Code files are filtered by removing the ones that have duplicates and any lines longer than 3000 tokens or average line length greater than 100 tokens

Training

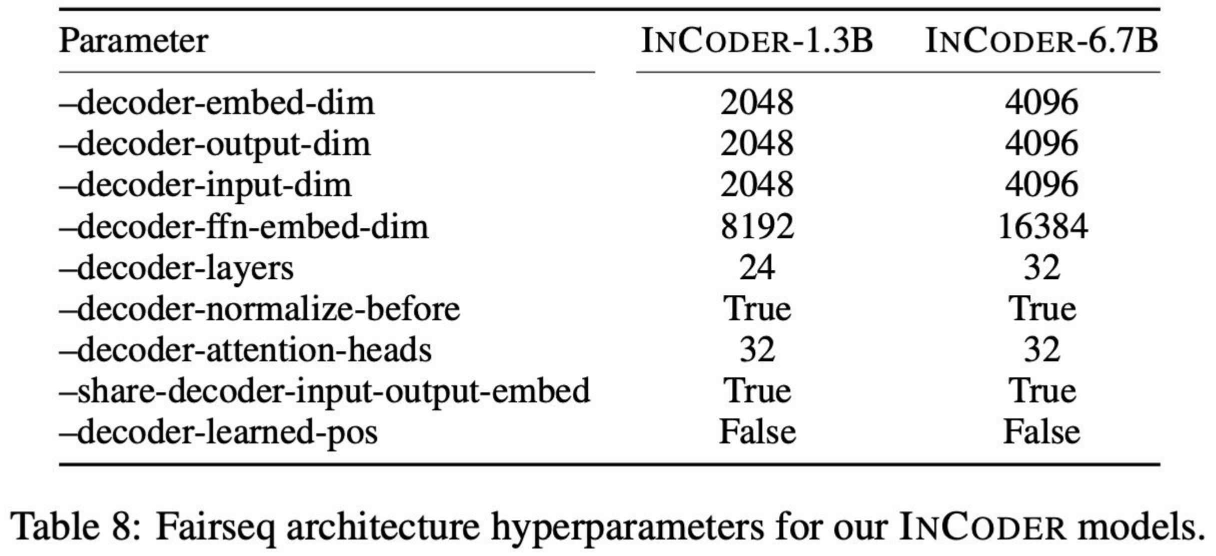
Metadata is also included for code files (file name, file extension, file source, number of stars) and for stackoverflow data (question tags and number of votes)

Code is tokenized by using byte-level BPE tokenizer

Tokens are allowed to extend over white spaces to represent certain programming idioms as one token like import numpy as np

Training

Model architecture is based on Fairseq architecture and trained on 248 V100 GPUs for 24 days with Adam optimizer and polynomially decaying learning rate



Parameter	INCODER-1.3B	INCODER-6.7B
–decoder-embed-dim	2048	4096
–decoder-output-dim	2048	4096
–decoder-input-dim	2048	4096
–decoder-ffn-embed-dim	8192	16384
–decoder-layers	24	32
–decoder-normalize-before	True	True
–decoder-attention-heads	32	32
–share-decoder-input-output-embed	True	True
–decoder-learned-pos	False	False

Table 8: Fairseq architecture hyperparameters for our INCODER models.

Training

Even after 24 days of training they claim that their model is not yet saturated

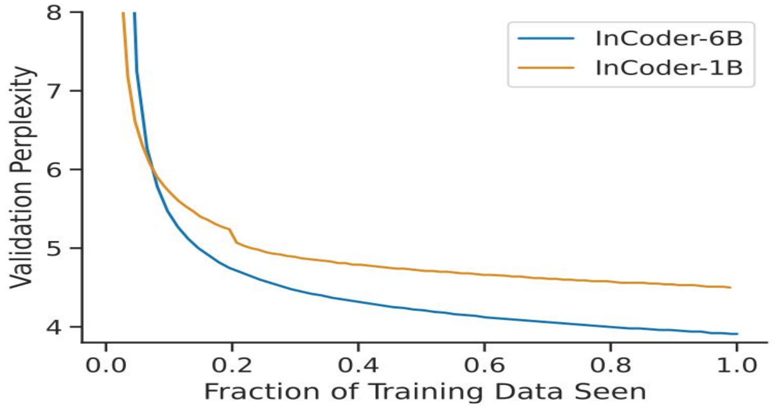


Figure 2: Loss curves show that perplexity is still improving after one epoch and that perplexity improves substantially with a larger model size. This suggests that increasing epochs, data size, or model size would improve performance.

Experiments

Mainly two types of experiments are performed: Infilling Experiments and Code Synthesis Experiments

Infilling Experiments are conducted to test the zero-shot infilling ability of the model on the following programming tasks : 1- Infilling Lines of Code 2- Docstring generation 3- Code Cloze 4- Return Type Prediction 5- Variable name prediction

Experiments

Compared their approach with two other techniques: standard left to right generation and left-to-right generation with reranking

Left-to-right generation with reranking essentially uses the left context to propose candidates to infill the blank and both right and left context to rank it

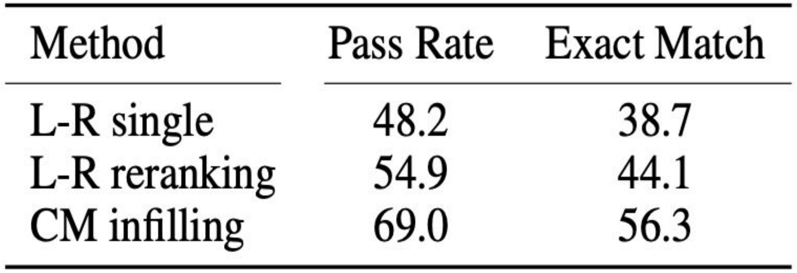
Infilling Lines of Code Results

Benchmarks are constructed from HumanEval

HumanEval has descriptions of functions paired with their implementation and several input–output test pairs

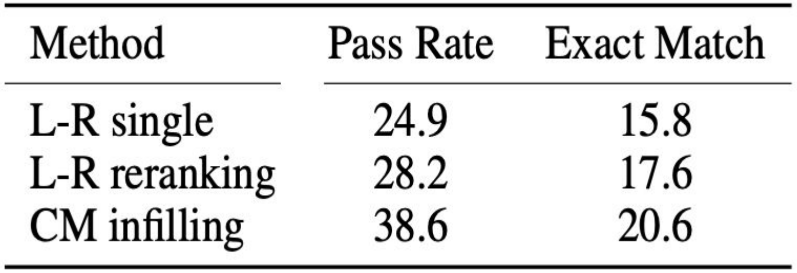
Both Multi-line infilling and Single-line infilling is analyzed using Exact Match and Pass rate (rate at which the completed function passes all of the function’s input–output pairs) metrics

Infilling Lines of Code Results



Method	Pass Rate	Exact Match
L-R single	48.2	38.7
L-R reranking	54.9	44.1
CM infilling	69.0	56.3

(a) Single-line infilling.



Method	Pass Rate	Exact Match
L-R single	24.9	15.8
L-R reranking	28.2	17.6
CM infilling	38.6	20.6

(b) Multi-line infilling.

Infilling Lines of Code Results

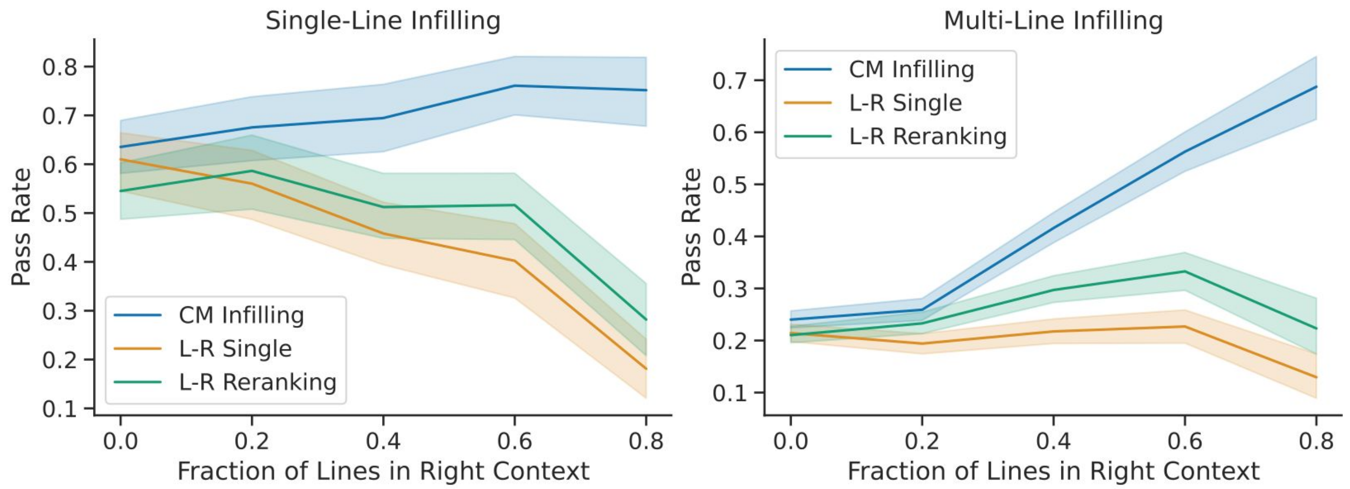
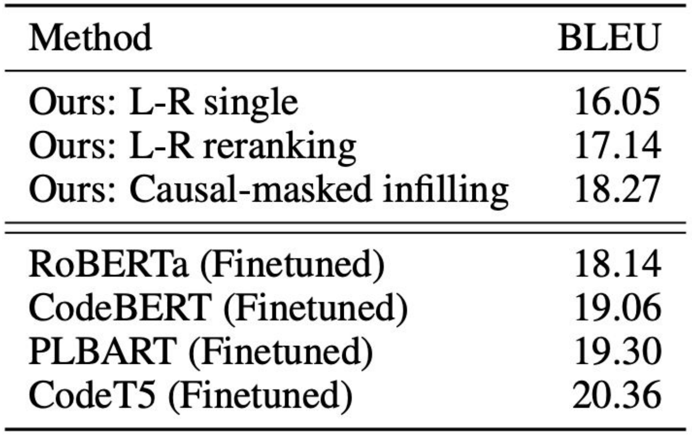


Figure 4: Infilling pass rate by the fraction of the function's lines which are provided to the right of the region that must be infilled, for single-line infilling (left) and multi-line infilling (right). Shaded regions give 95% confidence intervals, estimated using bootstrap resampling. Our causal-masked (CM) infilling method, blue, consistently outperforms both of the left-to-right (L-R) baselines, with larger gains as more right-sided context becomes available (the right side of both graphs).

Docstring Generation

CodexGLUE code to text benchmark is used (aim is to generate a natural language docstring that summarizes a Python code snippet)



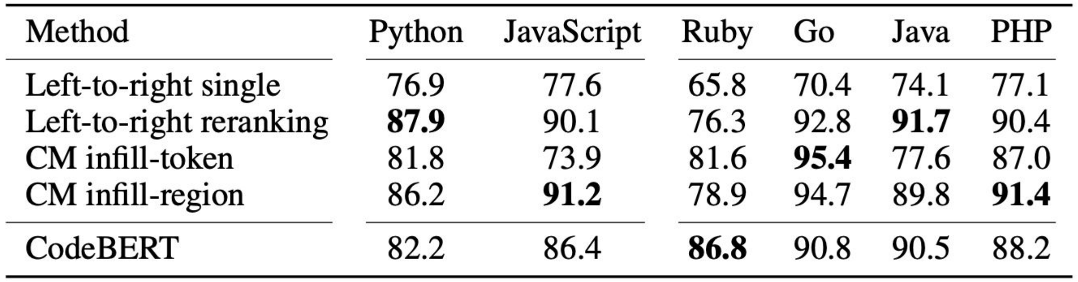
Method	BLEU
Ours: L-R single	16.05
Ours: L-R reranking	17.14
Ours: Causal-masked infilling	18.27
RoBERTa (Finetuned)	18.14
CodeBERT (Finetuned)	19.06
PLBART (Finetuned)	19.30
CodeT5 (Finetuned)	20.36

Table 2: CodeXGLUE Python Docstring generation BLEU scores. Our model is evaluated in a zero-shot setting, with no fine-tuning for docstring generation, but it approaches the performance of pretrained code models that are fine-tuned on the task's 250K examples (bottom block).

Code Cloze

CodexGLUE cloze set is used. (It is made of short natural language description along with their code in several programming languages)

Aim is to predict whether mask should be filled with max or min

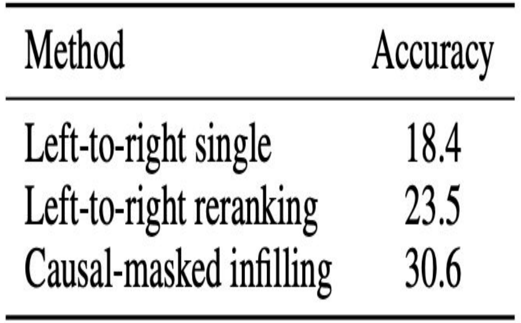


Method	Python	JavaScript	Ruby	Go	Java	PHP
Left-to-right single	76.9	77.6	65.8	70.4	74.1	77.1
Left-to-right reranking	**87.9**	90.1	76.3	92.8	**91.7**	90.4
CM infill-token	81.8	73.9	81.6	**95.4**	77.6	87.0
CM infill-region	86.2	**91.2**	78.9	94.7	89.8	**91.4**
CodeBERT	82.2	86.4	**86.8**	90.8	90.5	88.2

Table 3: Accuracy on the CodeXGLUE `max`/`min` cloze task. We compare four different inference methods. Left-to-right single: scoring with left-to-right ordering using only the left context and the completion (containing `max` or `min`); Left-to-right reranking: scoring with left-to-right ordering using the left context, completion, and right context; CM infill-token: causal masking scoring, using only a single token (containing `max` or `min`) as the infill, CM infill-region: causal masking scoring that additionally contains 10 tokens from the right side context.

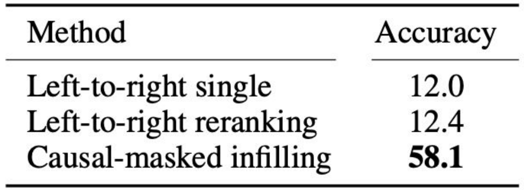
Return Type and Variable Name Prediction

Benchmarks are created from the CodexGlue



Method	Accuracy
Left-to-right single	18.4
Left-to-right reranking	23.5
Causal-masked infilling	30.6

Table 5: Results on the variable renaming benchmark that we construct from CodeXGLUE. Our model benefits from using the right-sided context in selecting (L-R reranking and CM infilling) and proposing (CM infilling) variable names.



Method	Accuracy
Left-to-right single	12.0
Left-to-right reranking	12.4
Causal-masked infilling	**58.1**

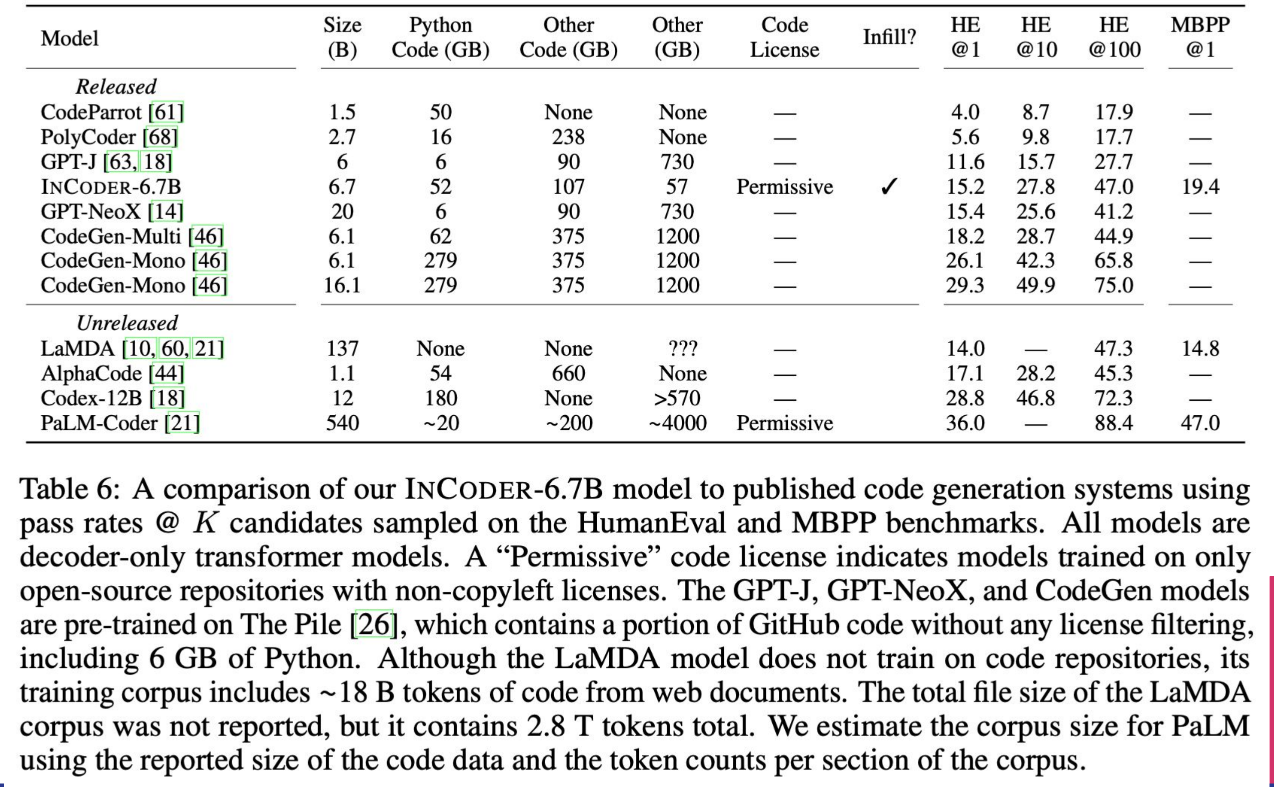
(a) Results on the test set of the benchmark that we construct from CodeXGLUE.

Code Synthesis

Evaluates zero shot performance of Incoder on Human Eval and MBPP benchmarks (aim is to condition on docstrings to generate python code)

Pass Rate on provided test suites is used as a evaluation metric

Code Synthesis



Model	Size (B)	Python Code (GB)	Other Code (GB)	Other (GB)	Code License	Infill?	HE @1	HE @10	HE @100	MBPP @1
Released										
CodeParrot [61]	1.5	50	None	None	—		4.0	8.7	17.9	—
PolyCoder [68]	2.7	16	238	None	—		5.6	9.8	17.7	—
GPT-J [63, 18]	6	6	90	730	—		11.6	15.7	27.7	—
INCODER-6.7B	6.7	52	107	57	Permissive	✓	15.2	27.8	47.0	19.4
GPT-NeoX [14]	20	6	90	730	—		15.4	25.6	41.2	—
CodeGen-Multi [46]	6.1	62	375	1200	—		18.2	28.7	44.9	—
CodeGen-Mono [46]	6.1	279	375	1200	—		26.1	42.3	65.8	—
CodeGen-Mono [46]	16.1	279	375	1200	—		29.3	49.9	75.0	—
Unreleased										
LaMDA [10, 60, 21]	137	None	None	???	—		14.0	—	47.3	14.8
AlphaCode [44]	1.1	54	660	None	—		17.1	28.2	45.3	—
Codex-12B [18]	12	180	None	>570	—		28.8	46.8	72.3	—
PaLM-Coder [21]	540	~20	~200	~4000	Permissive		36.0	—	88.4	47.0

Table 6: A comparison of our INCODER-6.7B model to published code generation systems using pass rates @ K candidates sampled on the HumanEval and MBPP benchmarks. All models are decoder-only transformer models. A "Permissive" code license indicates models trained on only open-source repositories with non-copyleft licenses. The GPT-J, GPT-NeoX, and CodeGen models are pre-trained on The Pile [26], which contains a portion of GitHub code without any license filtering, including 6 GB of Python. Although the LaMDA model does not train on code repositories, its training corpus includes ~18 B tokens of code from web documents. The total file size of the LaMDA corpus was not reported, but it contains 2.8 T tokens total. We estimate the corpus size for PaLM using the reported size of the code data and the token counts per section of the corpus.

Strengths

Proposes a novel casual masking approach that combines the strength of both MLM and causal models

It has strong zero-shot performance on most of the editing and code infilling and tasks

Provides functionality to edit any arbitrary region of the code which is often neglected by the other large scale transformer based models

Limitations

Mostly focuses on code editing infilling, code synthesis performance of the model is not that well explored

Full potential of the model is not measured since even after 24 days their validation performance was increasing

Future work: Supervised infilling via model fine-tuning and or iterative decoding with model refining its own output

References

Fried, Daniel, et al. "Incoder: A generative model for code infilling and synthesis." arXiv preprint arXiv:2204.05999 (2022).