This paper introduces CodeGemma, a collection of specialized open code models built on top of Gemma, capable of a variety of code and natural language generation tasks. We release three model checkpoints. CodeGemma 7B pretrained (PT) and instruction-tuned (IT) variants have remarkably resilient natural language understanding, excel in mathematical reasoning, and match code capabilities of other open models. CodeGemma 2B is a state-of-the-art code completion model designed for fast code infilling and open-ended generation in latency-sensitive settings.

Introduction

We present CodeGemma, a collection of open code models based on Google DeepMind’s Gemma models (Gemma Team et al., 2024).

Continuing from Gemma pretrained models, CodeGemma models are further trained on more than 500 billion tokens of primarily code, using the same architectures as the Gemma model family. As a result, CodeGemma models achieve state-of-the-art code performance in both completion and generation tasks, while maintaining strong understanding and reasoning skills at scale. We release a 7B code pretrained model and a 7B instruction-tuned code model. Further, we release a specialized 2B model, trained specifically for code infilling and open-ended generation. The lineage of these models is depicted in Figure 1.

In this report, we provide an overview of the additions to Gemma, such as pretraining and instruction-tuning details for CodeGemma, followed by evaluations of all models across a wide variety of academic and real world tasks against similar models. Finally, we outline the areas in which CodeGemma excels and its limitations, followed by recommendations for using this model.

Pretraining

Training Data

CodeGemma models are further trained on 500 billion tokens of primarily English language data from web documents, mathematics, and code. The 2B models are trained with 100% code while the 7B models are trained with a 80% code-20% natural language mixture. Our code corpus comes from publicly available code repositories. Datasets are deduplicated and filtered to remove contamination of evaluation code and certain personal and sensitive data. In addition to the processing done for Gemma, we perform additional pretraining steps for code data.

Preprocessing for Fill-in-the-Middle

The pretrained CodeGemma models are trained using a method based on the fill-in-the-middle (FIM) task (Bavarian et al., 2022) with improvements that address the shortcomings cited in the
original work as well as empirically-found systemic issues with existing FIM-trained models. The relevant formatting control tokens are presented in Table 1. The models are trained to work with both PSM (Prefix-Suffix-Middle) and SPM (Suffix-Prefix-Middle) modes. Figure 2 shows a sample snippet formatted in PSM. We make detailed FIM usage instructions in the Inference Recommendations section.

<table>
<thead>
<tr>
<th>Context</th>
<th>Relevant Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIM prefix</td>
<td>&lt;</td>
</tr>
<tr>
<td>FIM middle</td>
<td>&lt;</td>
</tr>
<tr>
<td>FIM suffix</td>
<td>&lt;</td>
</tr>
<tr>
<td>File separator</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

Table 1 | Formatting control tokens used for FIM task. Note that | is the standard pipe character (ASCII code 124).

Multi-file Packing

Many downstream code-related tasks involve generating code based on a repository-level context as opposed to a single file. To improve model alignment with real-world applications, we create training examples by co-locating the most relevant source files within code repositories and best-effort grouping them into the same training examples. Specifically, we employ two heuristics: dependency graph-based packing and unit test-based lexical packing.

To construct the dependency graph, we first group files by repository. For each source file, we extract imports from the top N lines and perform suffix matching to determine the longest matching paths within the repository structure. We determine edge importance (a heuristic measure) between files, and remove unimportant edges to break cyclic dependencies (common in Python). We then calculate all-pairs shortest paths within the graph, where shorter distances signify stronger file relationships. Finally, we linearize the graph of files using a topological sort, selecting the next unparented node based on minimum distance to sorted nodes and using lexicographic order to break ties.

Files not covered by this dependency graph method are sorted alphabetically within their repository with unit tests packed next to their implementations (e.g. TestFoo.java beside Foo.java).

Instruction Tuning

Our training data consists of a combination of open-source math datasets and synthetically generated code, in addition to the finetuning datasets used by Gemma. By exposing the model to mathematical problems, we aim to enhance its logical reasoning and problem-solving skills, which are essential for code generation.

Mathematics Datasets

To enhance the mathematical reasoning capabilities of coding models, we employ supervised fine-tuning on a diverse set of mathematics datasets, including:

**MATH Dataset** A collection of 12,500 challenging mathematical problems from competitions, providing step-by-step solutions for training models in answer derivation and explanation generation (Hendrycks et al., 2021).

**GSM8k Dataset** A collection of 8,500 grade school math problems. This dataset tests the multi-step reasoning abilities of models, highlighting their limitations despite the simplicity of the problems (Cobbe et al., 2021a).

**MathQA Dataset** A large-scale dataset of math word problems (Amini et al., 2019) with annotations built on top of the AQuA dataset (Ling et al., 2017).

**Synthetic Mathematical Data** A programmatically-generated dataset of algebraic problems used to improve ability to solve long algebra problems.

By leveraging these diverse datasets, we expose the model to a wide range of mathematical
def mean_absolute_deviation(numbers: List[float]) -> float:
    """For a given list of input numbers, calculate Mean Absolute Deviation around
    the mean of this dataset."
    Mean Absolute Deviation is the average absolute difference between each element
    and a centerpoint (mean in this case):
    MAD = average | x - x_mean |
    >>> mean_absolute_deviation([1.0, 2.0, 3.0, 4.0])
    1.0
    """
    return sum(abs(x - mean) for x in numbers) / len(numbers)

Figure 2 | Example code snippet in PSM mode. The green characters are part of the format, whereas uncolored is from the source. The shown code sample is from HumanEval (Chen et al., 2021).

problems, increasing their ability to perform complex mathematical reasoning. Our training experiments indicate that these datasets significantly boost code generation performance.

Coding Dataset

Effectively instruction-tuning large language models for code generation tasks requires a substantial amount of question-answer pairs. We leverage synthetic code instruction data generation to create datasets used in the supervised-finetuning (SFT) and reinforcement learning from human feedback (RLHF) phase. We apply the following steps:

Example Generation Following the approach outlined in the OSS-Instruct paper (Wei et al., 2023), we generate a set of self-contained question-answer pairs.

Post-Filtering We filter question-answer pairs using an LLM tasked with evaluating the helpfulness and correctness of the generated question-answer pairs.

Evaluation

We evaluate CodeGemma for code completion and generation performance, as well as natural language understanding, with automated benchmarks across a variety of domains.

Infilling Capability

HumanEval Infilling

The CodeGemma models are trained for code completion purposes. We use the single-line and multi-line metrics in the HumanEval Infilling benchmarks introduced in Fried et al. (2023) to evaluate. Performance against other FIM-aware code models is shown in Table 2.

We observe that our 2B pretrained model is an excellent well-rounded model for code completion use cases, where low latency is a critical factor. It performs on par with the other models while being, in many cases, nearly twice as fast during inference. We attribute this speedup to the base Gemma architectural decisions.

Real-world Evaluation

We validate our model’s infilling abilities by masking out random snippets in code with cross-file dependencies, generating samples from the model, and retesting the code files with the generated snippets to show that it performs as expected, a similar approach to Liu et al. (2023) or Ding et al. (2023). Due to our inclusion of very recently committed open source code, we do not use the evaluations directly, but use an internal version with the same testing methodology.

In addition to evaluating on offline evaluations,
<table>
<thead>
<tr>
<th>Model</th>
<th>Time (s)</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
<td>Multi</td>
</tr>
<tr>
<td><strong>2B class</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeGemma</td>
<td>543</td>
<td>8479</td>
</tr>
<tr>
<td>DeepSeek Coder</td>
<td>990</td>
<td>13138</td>
</tr>
<tr>
<td>DeepSeek Coder Instruct</td>
<td>5632</td>
<td>31505</td>
</tr>
<tr>
<td>StarCoder2</td>
<td>3665</td>
<td>20629</td>
</tr>
<tr>
<td>CodeGemma Instruct</td>
<td>8330</td>
<td>49438</td>
</tr>
<tr>
<td>Code Llama*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepSeek Coder</td>
<td>1559</td>
<td>22387</td>
</tr>
<tr>
<td>DeepSeek Coder Instruct</td>
<td>9500</td>
<td>53498</td>
</tr>
<tr>
<td>StarCoder2</td>
<td>8080</td>
<td>45459</td>
</tr>
</tbody>
</table>

Table 2 | Single-line and multi-line code completion capability of CodeGemma compared to other FIM-aware code models. Time is the total number of seconds to obtain 128-token continuations per each HumanEval Infilling task (1033 tasks in single-line and 5815 multi-line). Measurements are done with HuggingFace’s Transformers (Wolf et al., 2020) model implementations on g2-standard-4 GCE instances with bfloat16 datatype and batch size of 1. * Code Llama numbers are taken from Rozière et al. (2024).

the model was tested within live coding environments to benchmark its performance against current Google completion models.

**Coding Capability**

**Python Coding**

The canonical benchmarks used in coding evaluation are HumanEval (Chen et al., 2021) and Mostly Basic Python Problems (Austin et al., 2021). We present our results in Table 3.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HumanEval</th>
<th>MBPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2B-PT</td>
<td>31.1%</td>
<td>43.6%</td>
</tr>
<tr>
<td>Gemma 2B PT</td>
<td>22.0%</td>
<td>29.2%</td>
</tr>
<tr>
<td>7B-PT</td>
<td>44.5%</td>
<td>56.2%</td>
</tr>
<tr>
<td>7B-IT</td>
<td>56.1%</td>
<td>54.2%</td>
</tr>
<tr>
<td>Gemma 7B PT</td>
<td>32.3%</td>
<td>44.4%</td>
</tr>
</tbody>
</table>

Table 3 | Python coding capability of CodeGemma on de-facto coding benchmarks.

Compared to the base Gemma models (Gemma Team et al., 2024), CodeGemma models perform significantly better on tasks from the coding domain.

**Multi-lingual Benchmarks**

BabelCode (Orlanski et al., 2023) is used to measure the performance of CodeGemma on code generation across a variety of popular programming languages. Results are presented in Table 4.

**Language Capability**

We evaluate performance on a variety of domains including question answering (Bisk et al., 2019; Clark et al., 2019, 2018; Joshi et al., 2017), natural language (Hendrycks et al., 2020; Sakaguchi et al., 2019; Zellers et al., 2019) and mathematical reasoning (Gobbe et al., 2021b; Hendrycks et al., 2021). We present the results of our two 7B models next to the instruction-tuned Gemma 7B model in Figure 3.

CodeGemma retains most of the same natural language capabilities seen in the base Gemma models. CodeGemma PT and IT both outperform Mistral 7B (Jiang et al., 2023) by 7.2% and Llama-2 13B model (Touvron et al., 2023) by 19.1% (numbers reported in Gemma Team et al. 2024). Further, we compare scores for GSM8K and MATH in Table 5 from several code models in the 7B
Table 4 | Multi-lingual coding capability of CodeGemma (CG) on BabelCode-translated HumanEval and Mostly Basic Python Problems (MBPP) datasets. IT stands for instruction-tuned.

<table>
<thead>
<tr>
<th>Language</th>
<th>2B</th>
<th>7B</th>
<th>7B-IT</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>24.2%</td>
<td>32.9%</td>
<td>42.2%</td>
</tr>
<tr>
<td>C#</td>
<td>10.6%</td>
<td>22.4%</td>
<td>26.7%</td>
</tr>
<tr>
<td>Go</td>
<td>20.5%</td>
<td>21.7%</td>
<td>28.6%</td>
</tr>
<tr>
<td>Java</td>
<td>29.2%</td>
<td>41.0%</td>
<td>48.4%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>21.7%</td>
<td>39.8%</td>
<td>46.0%</td>
</tr>
<tr>
<td>Kotlin</td>
<td>28.0%</td>
<td>39.8%</td>
<td>51.6%</td>
</tr>
<tr>
<td>Python</td>
<td>21.7%</td>
<td>42.2%</td>
<td>48.4%</td>
</tr>
<tr>
<td>Rust</td>
<td>26.7%</td>
<td>34.1%</td>
<td>36.0%</td>
</tr>
</tbody>
</table>

Table 5 | Math reasoning capability of other code models in the same 7B size class. Results collected from Guo et al. (2024); Lozhkov et al. (2024); Rozière et al. (2024).

<table>
<thead>
<tr>
<th>Model</th>
<th>GSM8K</th>
<th>MATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeGemma PT</td>
<td>44.2%</td>
<td>19.9%</td>
</tr>
<tr>
<td>CodeGemma IT</td>
<td>41.2%</td>
<td>20.9%</td>
</tr>
<tr>
<td>Code Llama</td>
<td>13.0%</td>
<td></td>
</tr>
<tr>
<td>DeepSeek Coder</td>
<td>43.2%</td>
<td>19.2%</td>
</tr>
<tr>
<td>StarCoder2</td>
<td>40.4%</td>
<td></td>
</tr>
</tbody>
</table>

**Practical Considerations**

CodeGemma is tailored for practical use and deployment in latency-sensitive settings. The 2B model is considerably faster than all models in our comparison set, which is critical for latency-sensitive applications such as code completion. This speedup does not come with a significant, measured compromise in quality according to our evaluations — the 2B model performs as well or better compared to other open models in its class at code infilling tasks. Consequently, CodeGemma 2B is exceptionally suitable for utilization within Integrated Development Environments (IDEs), local environments, and other applications with memory constraints.

The 7B models, characterized by their strong performance, are general coding models that surpass the baseline Gemma models in terms of coding tasks while maintaining a high level of natural language comprehension. The larger memory requirement during inference renders these models particularly suitable for deployment in hosted environments and applications where model quality is of utmost importance.

The Responsible Deployment section in Gemma Team et al. (2024) contains a thorough discussion about the limitations and benefits of using an open model.

**Inference Recommendations**

For pretrained models, prompts should be formatted for code completion tasks such as function completion, docstring generation, and im-
port suggestion. Figure 4 shows an example of a prompt format, where the file path is optional but recommended. The stopping strategy for model outputs should be chosen carefully to align with the deployment setting. The most straightforward method is to truncate upon generating a FIM sentinel token, as shown in Table 1.

```python
path/file.py
<|fim_prefix|>prefix<|fim_suffix|>suffix
<|fim_middle|>
```

Figure 4 | Prompt in PSM mode. The carriage return is part of the format. There are no spaces after the suffix.

The same formatting as Gemma, with `<start_of_turn>` and `<end_of_turn>` tokens, can also prompt the instruction-tuned model.

**Conclusion**

We present a collection of open models specialized for coding applications, built on top of Gemma, an openly available family of language models (Gemma Team et al., 2024). These models push the state of the art in code completion and generation, while retaining natural language capabilities from the base models.

The CodeGemma models presented in this report are highly capable language models designed for effective real-world deployment, optimized to be run in latency-constrained settings while delivering high-quality code completion on a variety of tasks and languages. We show that the lessons and technologies that built Gemini and Gemma are transferable to downstream applications, and we are excited to release these models to the broader community and to enable the applications which will be built on top of these models.
Contributions and Acknowledgments

Core Contributors
赵赫 (Heri Zhao)
许嘉伦 (Jeffrey Hui)
Joshua Howland
Nguyễn Thanh Nam¹ (Nam Nguyen)
左斯琦 (Siqi Zuo)

Contributors
胡琪恩 (Andrea Hu)
Christopher A. Choquette-Choo
Jingyue Shen
Joe Kelley
Kshitij Bansal
Luke Vilnis
Mateo Wirth
Paul Michel
Peter Choy
Pratik Joshi
Sarmad Hashmi
Shubham Agrawal
Zhitao Gong

Product Management
Jane Fine
Tris Warkentin

Program Management
Ale Jakse Hartman

Executive Sponsors
Bin Ni
Kathy Korevec
Kelly Schaefer
Scott Huffman

Acknowledgements
Our work is made possible by the dedication and efforts of numerous teams at Google. We would like to acknowledge the support from the following teams: AIDA, DevRel, Gemini Infrastructure, Gemini Safety, Gemma, Google Cloud, Google Research Responsible AI, Kaggle, Keras.

Special thanks and acknowledgment to Alek Andreiev, आलिफ्लो श्रीराम (Anirudh Sriram), Antonia Paterson, अरोमा महेंद्र (Aroma Mahendra), Arthur Zucker, Austin Huang, David Huntsperger, ध्वनिक विरडिया (Dhvanik Viradiya), Elisa Bandy, Emma Yousif, गौरण्ग कोठिया (Gaurang Kothiya), Glenn Cameron, हेतुल पटेल (Hetal Patel), James Freedman, Jasmine George, Jenny Brennan, Johan Ferret, Josh Woodward, Kathleen Kenealy, Keelin McDonell, Lav Rai, Léonard Husseren, लोबना बेन अलाल (Loubna Ben Allal), Ludovic Peran, Luiz Gustavo Martin, Manvinder Singh, Matthew Watson, Meg Risdal, Michael Butler, Michael Moynihan, Min Kim (Minwoo Park), Minh Giang, Morgane Riviere, Navneet Potti, Nino Vieillard, Olivier Bachem, Omar Sanseviero, Pedro Cuenca, Phil Culliton, Pier Giuseppe Sessa, राज गुंडलुरु (Raj Gundluru), Robert Dadashi, संजना पुरोहित (Sanjana Purohit), Sertan Girgin, शृंखलाकेश (Surya Bhupatiraju), उत्कर्ष पांड्या (Utkarsh Pandya), 单志昊 (Zhihao Shan).

References


P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018.


