# Scaling Laws for Code: A More Data-Hungry Regime

**Anonymous authors**
Paper under double-blind review

## Abstract

The training of large language models (LLMs) for code generation incurs substantial computational costs, yet the resource allocation strategies are often guided by scaling laws derived from natural language (NL). Given the distinct statistical properties of code, it is unclear if these heuristics are optimal. This paper presents the first large-scale, systematic investigation into the scaling laws of LLMs trained specifically on source code. We conduct 117 training runs, spanning model sizes from 0.2B to 3.8B parameters and dataset sizes from 2B to 128B tokens, to derive a precise scaling law for code. Our findings show that while code models adhere to the existing Farseer scaling law paradigm, they operate in a fundamentally different, "more data-hungry" regime. The compute-optimal data-to-parameter (D/N) ratio for code is significantly higher than for NL and accelerates with the compute budget. This suggests that the primary bottleneck for state-of-the-art code models is data availability, not diminishing returns from model size. Furthermore, through two additional sets of 117 experiments on code-NL mixtures, we find that while adding NL data can benefit smaller models in low-data scenarios, pure in-domain data is superior for larger-scale, compute-optimal training. Our findings provide a more refined, practical guide for the compute-optimal pre-training of LLMs for code.

## 1 Introduction

Large-scale training on code corpora has enabled Code Large Language Models (LLMs) to achieve remarkable code-related capabilities (Chen et al., 2021; Li et al., 2022; Roziere et al., 2023; Li et al., 2023; OpenAI et al., 2025). Built upon these models, applications such as OS, GUI, and Terminal Agents significantly enhance developer productivity and substantially impact the field (Zheng et al.,

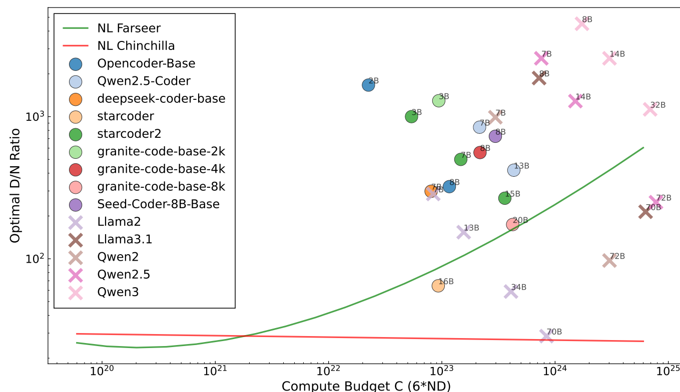


Figure 1: The relationship between compute budget (C) and the optimal loss for various existing language models. The Farseer and Chinchilla curves, derived from natural language data, diverge from the observed performance of several code-centric models, indicating that a different scaling behavior may govern code.

2024; Hong et al., 2024; Mei et al., 2024). The fuel for this technological revolution is the continuous growth of data and model size, which also incurs substantial computational costs (Kaplan et al., 2020; Hoffmann et al., 2022). Training frontier LLMs requires thousands of petaflop/s-days and millions of dollars, making it impractical to conduct ablation experiments on the largest models, whether on the structure, data, or training strategies.

Scaling Laws provide a theoretical foundation to address this challenge (Kaplan et al., 2020; Hoffmann et al., 2022). By using results from smaller models to fit empirical formulas, Scaling Laws describe the relationship between model performance and factors like model size (N), dataset size (D), and compute (C). They offer guidance for training general-purpose natural language (NL) LLMs and provide important references for resource allocation on the data and model. However, code, as a highly structured data type, has statistical properties that fundamentally differ from NL: it has strict syntax, complex long-range dependencies, and unique vocabulary distributions. This raises the question: *Can Scaling Laws developed for NL be applied to code?*

Exploring this open question is particularly important. On one hand, Code LLMs tend to be concentrated in smaller model sizes, unlike NL LLMs, which have a more even distribution of sizes. *Does model scaling for Code LLMs plateau more quickly?* On the other hand, Code LLMs often require a larger data size relative to their model size. *Does it mean that Code LLMs require more code data?* Answering these questions is crucial for understanding code pretraining and guiding the future development of large models for code.

To address these questions, we conduct the first systematic explorations of Scaling Laws on code. We perform 117 experiments with models from 0.2B to 3.8B parameters and datasets from 2B to 128B tokens to derive a code-specific scaling law. Our findings show that code adheres to the Farseer scaling paradigm without requiring a new formulation. We observe that Code LLMs scale robustly with model size but show diminishing returns with data size after a certain point. Further analysis of the compute-optimal data-to-parameter (D/N) ratio reveals that code is significantly more "data-hungry" than natural language (NL). For a given model size, a Code LLM requires substantially more data to reach its optimal performance. This suggests that the smaller size of typical code models is not due to a lack of benefit from scaling model parameters, but rather the scarcity of available high-quality code data.

Given the relative abundance of NL data, we also investigate its potential to augment code model training. We conduct two additional sets of 117 experiments using 70%/30% and 30%/70% code-NL mixtures. The results indicate that when the volume of code data is limited, incorporating a moderate amount of NL data can indeed enhance performance on code-related tasks. However, this benefit diminishes and eventually becomes a detriment as the proportion of NL data increases or as more pure code data becomes available.

Our contributions are summarized as follows:

- To our knowledge, we conduct one of the first Scaling Law studies on code data, showing that code adheres to the Scaling Law paradigm and can be well-predicted within our tested ranges.
- Scaling Laws demonstrate that Code LLMs exhibit good scaling properties with respect to model size. The compute optimal shows that the D/N ratio for code will be significantly larger than for NL, indicating that Code LLMs require more data for training.
- By adding two different NL-Code mixture ratios to our Scaling Law experiments, we show that when code data is limited, increasing NL data can indeed help enhance code performance.

# 2 METHODOLOGY

## 2.1 BACKGROUND AND MOTIVATION

Scaling laws are empirical formulas that guide the efficient training of large-scale models by describing the relationship between a model's performance (loss) and key factors like model size ($N$), dataset size ($D$), and compute ($C$). Foundational work for natural language (NL), such as the Chinchilla scaling law Hoffmann et al. (2022), proposed a power-law relationship where

model and data size are scaled proportionally: $L(N, D) = E + \frac{A}{N^a} + \frac{B}{D^b}$. More recent refinements like Farseer Li et al. (2025b) have introduced more complex formulations, such as $L(N, D) = e^{s \cdot N^q + S} + e^{B \cdot N^b + Q} \cdot D^{-e^{A \cdot N^a + E}}$, to capture the nuanced interplay where larger models learn more efficiently from data.

However, while these laws have proven effective for NL, their direct applicability to code is questionable. Code possesses fundamentally different statistical properties, including strict syntax and complex long-range dependencies, that distinguish it from NL. This theoretical difference is supported by preliminary observations; as shown in Figure 1, the performance of existing code models does not align with the optimal data-to-model size (D/N) ratios predicted by NL-derived scaling laws. This discrepancy highlights the need for a dedicated investigation to establish a scaling law tailored specifically for code, which can provide more accurate guidance for training state-of-the-art code LLMs.

### 2.2 Experimental Design

To investigate the scaling properties of code, we conducted a comprehensive set of experiments. We adopted a principled experimental design methodology inspired by the large-scale approach of Farseer Li et al. (2025b), focusing on maximizing the information gained from each training run within a feasible computational budget.

**Sampling Strategy** Given the significant computational cost of training, we selected 117 distinct configurations of model size ($N$) and dataset size ($D$) for our experiments. To ensure comprehensive coverage across different orders of magnitude, we employed a log-uniform sampling strategy across both dimensions. Our chosen points span model sizes from 0.2B to 3.8B parameters and training tokens from 2B to 128B. This strategic selection, illustrated in Figure 2, provides a cost-effective basis for mapping the loss surface.

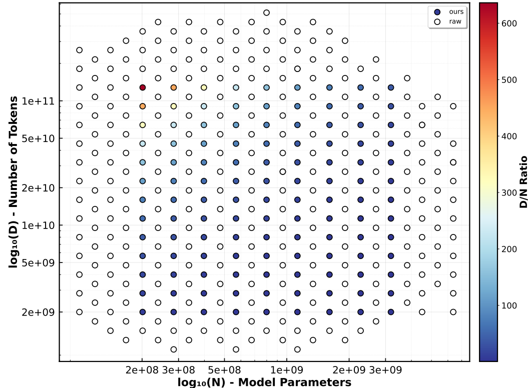


Figure 2: Scatter plot showing the relationship between model parameters (N) and training tokens (D) for our experimental configurations. Points are colored by D/N ratio, with ours (blue) representing our selected 117 training points and raw (white) showing all the Farseer experimental points.

**Model Architecture and Parameterization** To isolate the effects of scale, we maintained a consistent decoder-only Transformer architecture across all 117 models, incorporating standard components like SwiGLU activations, Rotary Position Embeddings (RoPE), and RMSNorm (Shazeer, 2020; Su et al., 2021; Zhang & Sennrich, 2019). We used a deterministic procedure, guided by the principles in Farseer Li et al. (2025b), to map a target parameter count $N$ to concrete architectural hyperparameters. This ensures that models at different scales are structurally sound and comparable by maintaining near-optimal aspect ratios. A detailed table of representative model configurations is provided in Appendix A.1.

**Training Hyperparameters** To ensure that each model was trained efficiently without a cost-prohibitive hyperparameter search, we followed StepLaw's optimal hyperparameter scaling rules to

set near-optimal learning rates, global batch sizes, and schedules as a function of model size $N$ (Li et al., 2025a). Notably, StepLaw reports and validates a code pretraining recipe (among diverse data recipes), which makes these settings appropriate for our code-focused study. We used the AdamW optimizer (Loshchilov & Hutter, 2019) with a cosine learning rate decay schedule for all experiments.

**Train Set and Validation Set** We train on a large, deduplicated public code corpus (895.51B tokens after sampling) and evaluate on a high-quality, closed-source code validation set (6.3M tokens). Full dataset composition, sampling ratios, and contamination checks are provided in Appendix A.2 (Table 3).

# 3 Experiment

## 3.1 Computational Setup

All experiments run on NVIDIA H100-80GB GPUs. We derive near-optimal global batch size and micro-batch size (MBZ) from $N$ and $D$ using compute-aware heuristics (consistent with StepLaw scaling rules), and set per-GPU MBZ by memory and sequence length. We use gradient accumulation to reach the target GBZ and choose the number of GPUs (8–128) to balance wall-clock time against MFU. The total compute expended is approximately 13,600 A100 GPU-days; further hardware and scheduling details are in Appendix A.3.

## 3.2 Scaling Behavior

After training, the validation loss is computed for each of the 117 models. As a prerequisite for fitting a scaling law, we first qualitatively verify that the model performance scales predictably with $N$ and $D$. Figure 3 provides a comprehensive visualization of this behavior. Figure 3(a) shows that for a fixed data budget, the validation loss monotonically decreases as $N$ increases, a trend that holds consistently across all tested data budgets. Symmetrically, Figure 3(b) demonstrates that for any given model size, the loss also smoothly declines as the number of $D$ grows. Both subplots are shown on double-logarithmic axes; the near-linear trends indicate approximate power-law relationships between loss and $N/D$, consistent with scaling-law assumptions. These clear and consistent trends suggest that performance on code exhibits stable scaling behavior, thus providing an empirical foundation for our subsequent quantitative analysis.

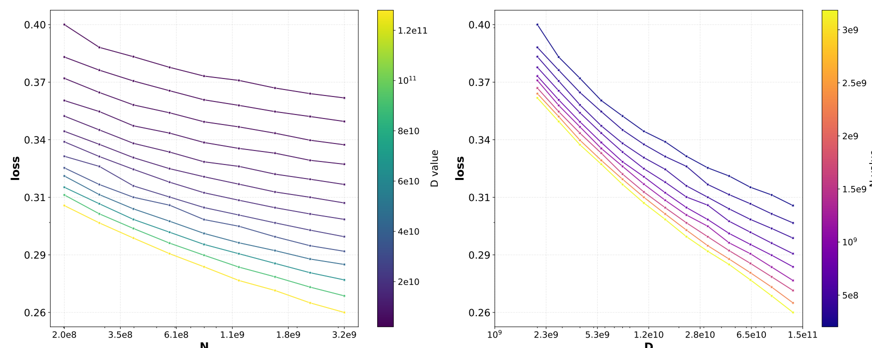


Figure 3: Scaling behavior of loss across different model sizes (N) and data sizes (D). (a) shows loss decreasing with increasing N for fixed D values, (b) shows loss decreasing with increasing D for fixed N values. The color gradients represent different D values (left) and N values (right). Both subplots are displayed on double-logarithmic axes; the near-linear trends indicate approximate power-law relationships between code-model loss and N/D under fixed D/N, respectively.

216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269

### 3.3 CODE SCALING LAW FITTING

To quantitatively model the observed scaling behavior, we fit both the Chinchilla and Farseer formulations to the complete set of our 117 experimental data points. The resulting fitted Chinchilla law is presented in Equation 1, and the Farseer law in Equation 2:

$$L(N, D) = 0.2193 + \frac{534.374}{N^{0.4853}} + \frac{76.0743}{D^{0.2983}} \quad (1)$$

$$\begin{aligned} L(N, D) = &\exp(-0.0047 \cdot N^{0.239} - 0.8188) \\ &+ \exp(62.8936 \cdot N^{-0.0614} - 14.0414) \cdot D^{-\exp(-0.0209 \cdot N^{0.1943} - 0.1826)} \end{aligned} \quad (2)$$

For asymptotic (irreducible) loss limits implied by these forms, please see Appendix A.4.

Farseer achieves a lower mean relative fitting error than Chinchilla (0.82‰vs. 1.03‰), suggesting it more accurately captures the underlying scaling dynamics of the observed data. However, the ultimate test of a scaling law is not merely its ability to fit existing data, but its power to extrapolate and predict performance in new, large-scale scenarios. Therefore, to comparatively assess their predictive utility, we design a dedicated validation experiment.

| N (B) | D (B) | D/N | GBZ | GPUs | MBZ | $PL_F$ | $PL_C$ | Loss | $RE_F$(‰) | $RE_C$(‰) |
|---|---|---|---|---|---|---|---|---|---|---|
| 6.37 | 127 | 20 | 640 | 160 | 2 | 0.259271 | 0.265707 | 0.256833 | **9.49** | 34.55 |
| 2.27 | 341 | 150 | 1080 | 120 | 9 | 0.253488 | 0.262330 | 0.253786 | **1.17** | 33.67 |
| 1.34 | 567 | 424 | 1456 | 112 | 13 | 0.255846 | 0.262939 | 0.258546 | **10.44** | 16.99 |

Table 1: Validation results for three model configurations trained at a fixed compute budget of $C = 5.36 \times 10^{21}$FLOPs. The selected D/N ratios represent the Chinchilla-prescribed optimum for natural language (20), the Farseer-predicted optimum for code (150), and an additional data-heavy validation point (424). **GPUs** indicates the number of GPUs used for the run. **GBZ** and **MBZ** refer to the Global and Micro Batch Size. $\mathbf{PL_F}$ and $\mathbf{RE_F}$ denote the predicted loss and relative error from Farseer, respectively, with $\mathbf{PL_C}$ and $\mathbf{RE_C}$ representing the counterparts for Chinchilla. **Loss** is the empirically measured validation loss.

To validate our fitted laws, we conducted experiments at a fixed compute budget of $C = 5.36 \times 10^{21}$FLOPs, comparing models at different D/N ratios as shown in Table 1. The empirical results yield two key findings. First, the NL-optimal ratio (D/N=20) is suboptimal for code, as a model trained at our predicted optimum (D/N=150) achieved significantly lower loss. Second, the Farseer law demonstrated superior predictive accuracy over Chinchilla across all validation points, with a remarkably low relative error of just 1.17‰at the empirically confirmed optimum. Farseer's accuracy, especially at the optimal point, confirms its effectiveness for guiding resource allocation for code model training.

In summary, our experiments demonstrate that while NL training recipes are suboptimal for code, code performance is nonetheless highly predictable under a scaling law paradigm. The existing Farseer formulation accurately captures these scaling dynamics without requiring a novel functional form, showing its adaptability to the code domain. We emphasize that our primary goal is to identify the most suitable *form* of the scaling law for code, not to establish universal constants for its parameters, as these may vary with different experimental setups.

## 4 ANALYSIS

### 4.1 SCALING SURFACE

Figure 4 summarizes two key views: the fitted Farseer surfaces for Code vs. NL over $(N, D)$, and the compute-optimal $D/N$ as a function of compute. Together they show that code attains lower loss across the plane and prefers higher $D/N$.

**Code is More Predictable than NL**. The validation losses for code and NL models are computed on their own validation sets. In our fitted surfaces and validation sets, code loss is generally lower

than NL across a broad range of $(N, D)$. This suggests a difference in intrinsic entropy between the two modalities. Although code can be logically complex, its statistical regularities—strict syntax, standard templates, keywords, and common programming idioms—are often more prevalent than in NL. This view aligns with evidence from speculative decoding, where code tokens are easier to predict than NL tokens, enabling higher acceleration (Leviathan et al., 2023). Differences across programming languages also appear smaller than across natural languages, making code tokens easier to learn and predict.

**Code LLMs Scale More Efficiently**. The Code surface exhibits a steeper overall gradient than NL. Investing additional compute into scaling either $N$ or $D$ yields larger marginal improvements for code. This may reflect a more focused domain with stronger transferability and generalization of patterns once syntax and structural priors are learned.

**Code Has Different Scaling Priorities.** Contrary to our initial hypothesis that code would plateau faster along $N$, we observe excellent scaling along the model-size axis: gains from increasing $N$ are larger for code than NL. Along the data-size axis, however, NL improves more consistently, whereas code saturates sooner. This is consistent with higher redundancy in large code corpora (e.g., GitHub), where additional tokens increasingly repeat known patterns; comparable gains require substantially more diverse data.

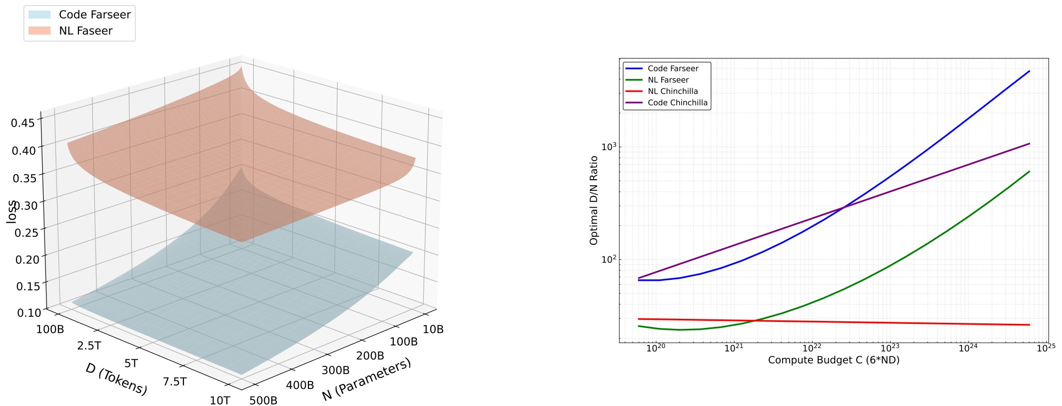


Figure 4: 3D visualization of the fitted Farseer scaling-law surfaces for Code (blue) and Natural Language (orange) over $(N, D)$ (left), and the compute-optimal $D/N$ ratio vs. compute budget $C$ (right). Code exhibits lower loss and steeper gradients across $(N, D)$, and a higher, accelerating optimal $D/N$ compared to NL in our fits.

### 4.2 Optimal D/N ratio

Based on the scaling landscapes, we derive the optimal D/N ratio as a function of compute budget C. For these calculations, we assume the FLOPs-per-token to be approximately 6.

**Code LLMs Tend to Be More Data-Hungry**. Across all compute budgets, the optimal D/N ratio for code is markedly higher than for NL, and this gap widens as compute increases. This suggests that achieving optimal performance for a Code LLM requires proportionally more data. We hypothesize this is due to the inherent repetitiveness of code; in later training stages, a much larger dataset is needed to provide novel learning signals. Consequently, targeted upsampling of high-quality, complex code may be more effective than simply expanding the corpus with common code.

**The Optimal D/N Ratio Grows with C, and May Accelerate**. The optimal D/N ratio for code is not static but grows with the compute budget (C). Our Chinchilla-style fit shows a linear increase, but the more accurate Farseer formulation exhibits super-linear growth. This acceleration in the Farseer model stems from the principle that larger models (N) learn from data more efficiently, thus demanding a larger dataset (D) for optimal training. This suggests that data becomes disproportionately more valuable at higher compute scales.

**D/N May Help Explain Smaller SOTA Code LLMs**. This high data requirement helps explain why mainstream code LLMs are generally smaller than their NL counterparts. The primary bottleneck appears to be the lack of sufficient data to optimally train a massive model, rather than diminishing returns from scaling the model size itself. In practice, model size is further constrained by the limited availability of code data and by factors like inference cost, especially for low-latency applications like code completion. Nevertheless, larger code LLMs can still deliver significant performance benefits when sufficient data and compute are available.

### 4.3 Data Mixing

Our analysis in Section 4.2 revealed that Code LLMs are significantly more data-hungry than their NL counterparts, a finding that presents a practical challenge given the relative scarcity of high-quality code data. This raises a critical question: can the vast corpora of NL be leveraged to improve code model performance? To investigate this, we conducted two additional large-scale experiments by creating distinct training datasets with the following mixture ratios:

- **70% Code + 30% NL**: A code-dominant mixture, denoted as `code_nl_73`.
- **30% Code + 70% NL**: An NL-dominant mixture, denoted as `code_nl_37`.

For each mixture, we replicated our full set of 117 experimental runs, covering the same range of N and D as the baseline. This rigorous approach allowed us to fit a new, distinct Farseer scaling law for each condition, enabling a direct and fair comparison of their scaling behaviors on our held-out code validation set.

The initial results of these experiments are summarized by the 3D loss surfaces in Figure 5. At a high level, the surface for the 100% code model (baseline) is generally the lowest across the studied (N, D) space, indicating stronger overall performance in our setting.

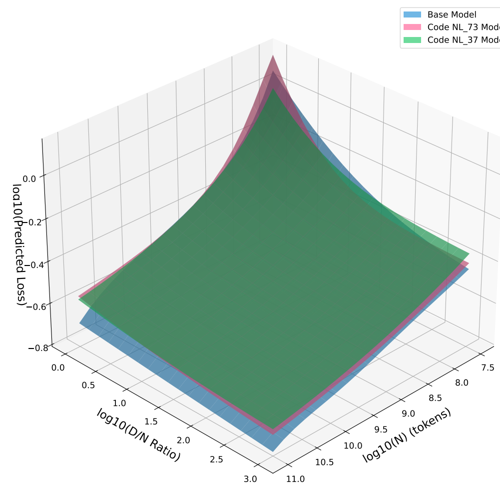


Figure 5: 3D visualization comparing scaling surfaces for different code-NL data mixtures: baseline (100% code), code_nl_73 (70% code + 30% NL), and code_nl_37 (30% code + 70% NL). In our setting, pure code typically achieves lower loss.

The 30/70 NL-dominant mixture (code_nl_37, green) consistently yields the highest (worst) loss. However, this holistic view masks a more intricate interaction, particularly between the baseline and the 70/30 mixture (code_nl_73, red). This interaction becomes clear when we analyze 2D slices of these surfaces.

Figure 6 plots the predicted BPC against the D/N ratio for several fixed model sizes (N). Here we observe a distinct crossover phenomenon for smaller models. For instance, with a model size of $N = 2.64 \times 10^8$, the code_nl_73 mixture (red dashed line) achieves a lower loss than the pure code model (blue solid line) in the low-data regime (approximately $D/N < 10$). This empirically

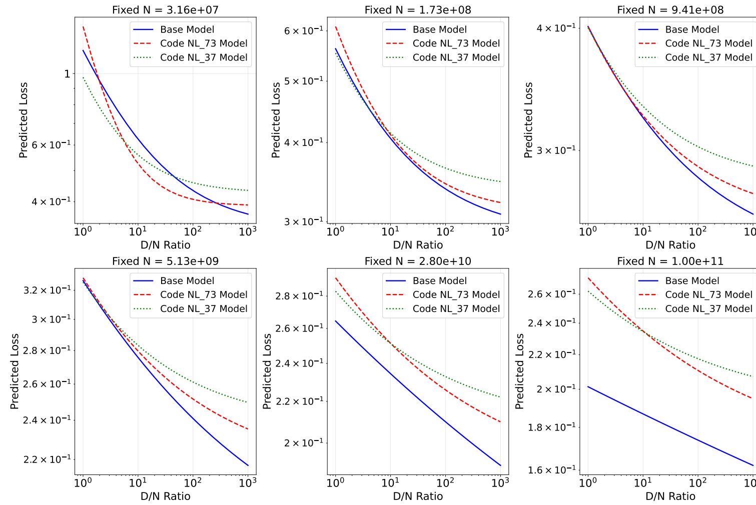


Figure 6: Predicted loss vs. D/N ratio for fixed model sizes. A crossover phenomenon is observed for smaller models: the 70/30 code-NL mixture performs better in low-data regimes, but pure code models excel with sufficient data. This benefit diminishes for larger models.

validates that when code data is limited, a moderate amount of NL data can indeed enhance a smaller model's performance on code tasks. However, as the D/N ratio increases, the pure code model's loss decreases more steeply, eventually surpassing the mixed model to become the superior choice.

This benefit of data mixing diminishes and ultimately disappears as model size increases. For large models (e.g., $N = 1.83 \times 10^{10}$; extrapolated), the fitted surfaces suggest that the pure code model performs better across a broad spectrum of D/N ratios. For these high-capacity models, the potential benefit of knowledge transfer from NL may be outweighed by the cost of distributional shift; the introduction of out-of-domain data can act as a performance impediment from the outset.

A complementary perspective is offered in Figure 7, which shows performance scaling with $N$ at several fixed D/N ratios. This view simulates a common scaling strategy where model size and data volume are increased proportionally.

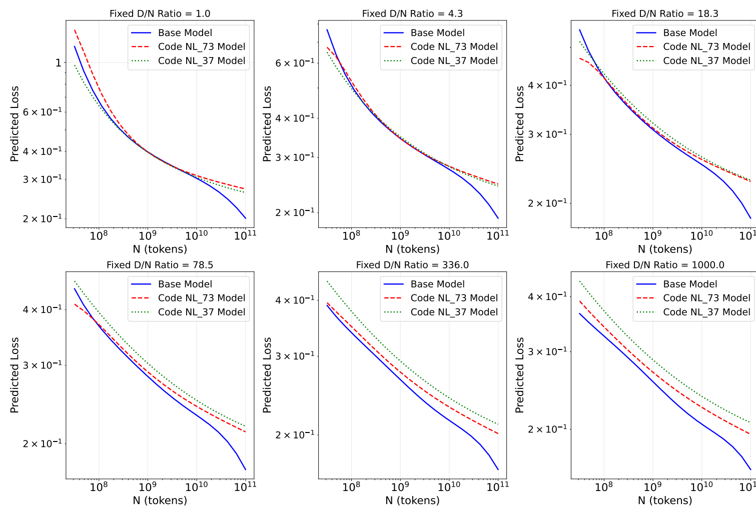


Figure 7: Predicted loss vs. model size ($N$) under a fixed D/N ratio scaling strategy. Each subplot corresponds to a constant D/N ratio. In this scaling scenario, the 100% code model (Base Model) generally outperforms the data mixtures across the tested ratios and model scales in our fitted models.

Unlike the crossover phenomenon observed when fixing model size, this scenario shows a clear trend: for a given D/N ratio, the 100% code model generally yields the lowest loss across model sizes within our tested ranges. This supports the view that if the goal is compute-optimal scaling prioritizing in-domain data purity appears effective in our setting from the outset. The benefit of mixing data appears to be confined to scenarios where a small model is constrained to an extremely low-data regime, rather than being a feature of an optimal scaling path.

We hypothesize two primary factors explain this behavior. The initial benefit for small models in low-data settings suggests that NL data provides valuable, transferable linguistic and structural priors. It helps the model build a more robust semantic understanding of concepts often described in code comments and variable names, which is particularly effective when code examples are sparse. However, as model capacity ($N$) and data volume ($D$) grow, the model becomes increasingly sensitive to the statistical purity of the training distribution. The model's vast capacity is more efficiently utilized by focusing solely on the intricate syntax and long-range structural patterns unique to code, making the inclusion of out-of-domain NL data a detrimental distraction.

## 5 Related Work

**Neural Scaling Laws.** The predictable scaling of Large Language Model (LLM) performance with compute, model size ($N$), and data volume ($D$) has been a cornerstone of their development. Early empirical studies established power-law relationships, advocating for a parameter-centric scaling strategy where model size was the primary focus (Kaplan et al., 2020). A significant paradigm shift occurred with the introduction of compute-optimal scaling laws, which demonstrated that for a fixed compute budget, model and dataset size should be scaled in tandem (Hoffmann et al., 2022). The Chinchilla laws suggested a near-constant optimal ratio of approximately 20 tokens per parameter. However, more recent work has refined this understanding, showing that the optimal $D/N$ ratio is not fixed but increases with the total compute budget. This dynamic is better captured by more complex functional forms, such as the Farseer law, which model the interdependence of $N$ and $D$ on the final loss and offer superior predictive accuracy for modern LLM training regimes (Li et al., 2025b).

**Large Language Models for Code.** Parallel to the development of general scaling laws, a distinct research area has focused on specializing LLMs for source code. This was pioneered by proprietary models like OpenAI's Codex, which set the standard for function-level synthesis (Chen et al., 2021), and DeepMind's AlphaCode, which demonstrated competitive performance on complex algorithmic reasoning tasks (Li et al., 2022). The success of these systems spurred the creation of powerful open-source alternatives, such as StarCoder (Li et al., 2023) and Code Llama (Roziere et al., 2023), which democratized access to state-of-the-art code intelligence. The current trend in the field emphasizes methodological transparency and reproducibility, as exemplified by projects like OpenCoder, which publish entire data processing and training methodologies (Huang et al., 2024).

Despite these advances, the specific scaling laws for LLMs trained on source code remain poorly understood. Consequently, training recipes often rely on heuristics or extrapolate from laws developed for natural language, a practice whose validity is unverified. Current code models are often trained on heuristic mixtures of code and natural language data without a strong empirical basis. This work addresses this gap by conducting the first large-scale, systematic study of scaling laws for code. We analyze models trained on pure code and controlled code-NL mixtures to establish a principled guide for the compute-optimal training of foundation models for code.

## 6 Conclusion

In this work, we present the first large-scale empirical study of code-specific scaling laws. Our results show that code follows the Farseer paradigm but in a distinctly more data-hungry regime: the compute-optimal data-to-parameter ($D/N$) ratio is significantly higher than for natural language and increases with the compute budget, empirically demonstrating that data availability is the primary bottleneck for improving code LLMs, not diminishing returns from model size. We further find that while mixing in natural language can benefit smaller models in low-data regimes, pure in-domain code is the superior choice for compute-optimal scaling. While our fitted parameters are specific

to our experimental setup, these findings provide concrete guidance for training future code LLMs and highlight promising directions for future work, such as validating these laws at larger scales and exploring more effective data curation strategies.

## ETHICAL STATEMENT

The data for the proposed methods is drawn solely from publicly accessible project resources on reputable websites, ensuring that no sensitive information is included. Moreover, all datasets and baseline models used in our experiments are taken care to acknowledge the original authors by properly citing their work.

594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647

# REFERENCES

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, Harrison Edwards, Christopher Hallacy, Kyla Wold, Yuchen He, Ryne Est-Sadam, Andrew Gibiansky, Kevin G. Lo, David Dohan, A. Power, Matthias Plappert, Heidy Khlaaf, Menal El-Maliki, David Luan, Girish Sastry, Chelsea Voss, Ingmar Kanitscheider, Jonathan Raiman, Kyle Kosic, Jan Leike, John Schulman, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *ArXiv*, abs/2203.15556, 2022.

Wenyi Hong, Weihan Wang, Qingsong Zhang, Zhipu AI, Zewen Chi, Li Dong, Xijie Wang, Wen-Hao Chen, Xing-Yu Ma, Yutao Sun, et al. Cogagent: A visual language model for gui agents. In *CVPR*, 2024.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. Opencoder: The open cookbook for top-tier code large language models. *ArXiv*, abs/2411.04905, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *ArXiv*, abs/2001.08361, 2020.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*, 2023.

Houyi Li, Wenzheng Zheng, Qiufeng Wang, Hanshan Zhang, Zili Wang, Shijie Xuyang, Yuantao Fan, Zhenyu Ding, Haoying Wang, Ning Ding, Shuigeng Zhou, Xiangyu Zhang, and Daxin Jiang. Predictable scale: Part i, step law – optimal hyperparameter scaling law in large language model pretraining. 2025a. URL `https://api.semanticscholar.org/CorpusID:280133901`.

Houyi Li et al. Predictable scale: Part ii, farseer: A refined scaling law in large language models. *ArXiv*, abs/2506.10972, 2025b.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Joao G. M. Araujo, G'erard Dupont, Daniel S. Fried, Dzmitry Bahdanau, Jian-Guo de Jesus, N. M. Marques, Yacine Jernite, Carlos Munoz Ferrandis, Sean Hughes, Arjun Guha, B. Chen, D. B. D. de Souza, A. F. P. de Paiva, Z. Szasz, Mayank Mishra, Alex Gu, Manan Dey, Logeswaran V, S. K. J. al., H. Wang, S. Singh, S. Bubeck, P. Y. Kocak, G. L. de Jesus, I. V. Serikov, N. T. Le, H. van der Veen, I. Sultan, G. E. L. P. Cost, S. Chim, T. T. Nguyen, M. T. Nguyen, T. Le, J. Sinclair, S. Gur, S. Baskaran, H. Le, H. Benyamina, D. A. Nguyen, M. S. Nguyen, M. D. Bui, P. H. Nguyen, N. D. Nguyen, T. T. H. Nguyen, V. D. Nguyen, L. M. C. Bui, H. T. Nguyen, N. T. Nguyen, A. H. Nguyen, T. T. Nguyen, H. V. Nguyen, V. H. Nguyen, L. T. Nguyen, H. H. Nguyen, N. P. Nguyen, N. M. Nguyen, V. H. Nguyen, D. V. Nguyen, D. Nguyen, T. Nguyen, M. Nguyen, T. Nguyen, H. Nguyen, T. Nguyen, T. Nguyen, H. Nguyen, T. T. Nguyen, T. T. Nguyen, T. T. Nguyen, H. V. Nguyen, V. H. Nguyen, L. T. Nguyen, H. H. Nguyen, N. P. Nguyen, N. M. Nguyen, V. H. Nguyen, D. V. Nguyen, D. Nguyen, T. Nguyen, M. Nguyen, T. Nguyen, and H. D. Vries. Starcoder: may the source be with you! *ArXiv*, abs/2305.06161, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, R'emi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven

Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378:1092–1097, 2022.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Tianyang Liu, Misha Laskin, et al. Starcoder2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971*, 2024.

OpenAI, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive Programming with Large Reasoning Models, 2025. URL https://arxiv.org/abs/2502.06807.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, J'er'emy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024.

702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755

# A APPENDIX

## A.1 MODEL CONFIGURATIONS

The architectural hyperparameters for a representative set of the 117 models are enumerated in Table 2. In accordance with the methodology outlined in Section 2.3, each model's architecture was algorithmically derived from its target non-embedding parameter count ($N$). This deterministic mapping maintains structural consistency and near-optimal aspect ratios across scales, thereby isolating the effects of scale from architectural variance.

Table 2: Architectural specifications for a representative subset of the models used in our scaling law experiments. All models share a consistent decoder-only Transformer architecture. $N$: Number of non-embedding parameters. $D_{\textbf{count}}$: The number of distinct dataset sizes this model architecture was trained on. $D_{\textbf{range}}$: The range of training tokens used for this architecture. $d_{\textbf{model}}$: The hidden dimension of the model. $d_{\textbf{ff}}$: The intermediate dimension of the feed-forward network. $N_{\textbf{head}}$: The number of attention heads. $N_{\textbf{layer}}$: The number of Transformer layers. $N_{\textbf{with_emb}}$: The total number of model parameters, including token embeddings.

| Model | $N$ | $D_{\text{count}}$ | $D_{\text{range}}$ | $d_{\text{model}}$ | $d_{\text{ff}}$ | $N_{\text{head}}$ | $N_{\text{layer}}$ | $N_{\text{with_emb}}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 201M | 13 | [2B, 128B] | 1024 | 2728 | 16 | 16 | 335M |
| 2 | 284M | 13 | [2B, 128B] | 1152 | 3032 | 18 | 18 | 435M |
| 3 | 398M | 13 | [2B, 128B] | 1280 | 3472 | 20 | 20 | 566M |
| 4 | 568M | 13 | [2B, 128B] | 1472 | 3888 | 23 | 22 | 761M |
| 5 | 798M | 13 | [2B, 91B] | 1600 | 4264 | 25 | 26 | 1.01B |
| 6 | 1.13B | 13 | [2B, 128B] | 1792 | 4832 | 28 | 29 | 1.36B |
| 7 | 1.61B | 13 | [2B, 128B] | 2048 | 5448 | 32 | 32 | 1.88B |
| 8 | 2.27B | 13 | [2B, 128B] | 2304 | 6064 | 36 | 36 | 2.58B |
| 9 | 3.18B | 13 | [4B, 128B] | 2560 | 6952 | 40 | 40 | 3.52B |

## A.2 DATASET AND VALIDATION DETAILS

We use the pretraining corpus released by OpenCoder (Huang et al., 2024) as the training set, which covers public GitHub content up to November 2023. The dataset has already undergone strict deduplication and rule-based filtering, following best practices established for large-scale open code corpora such as The Stack (Lozhkov et al., 2024). The strong performance of OpenCoder indicates the coverage and quality of this corpus. We remove synthetic data to better match the natural distribution of real-world code. After tokenization with our in-house tokenizer, the raw corpus contains 1216.93B tokens. Since knowledge density does not scale proportionally with raw volume across programming languages, we downsample languages with high redundancy and extreme volume to improve balance and pretraining efficiency. The resulting train set contains 895.51B tokens. Table 3 reports the detailed language composition and summary statistics. For the scaling-law experiments, training sets at different $D$ are obtained as nested subsets of the same pool. We shuffle the entire pool once with a fixed random seed and take progressively larger contiguous prefixes to form each budget. This design keeps the distribution stable across budgets. To evaluate trained models, we measure loss on a high-quality validation set drawn from a closed-source internal codebase. It is written by experienced engineers for production use, which aims to ensure realism and quality. Its closed-source nature reduces the risk of contamination from the public training pool. We further perform exact and near-duplicate checks against the training pool to guard against accidental overlap. Although no single source can perfectly represent the entire distribution of code, this validation set provides a consistent benchmark with checks to minimize contamination across all models. The final validation set contains 6.3M tokens.

## A.3 COMPUTATIONAL SETUP (DETAILS)

All experiments are conducted on a cluster of NVIDIA H100-80GB GPUs. A meticulous resource allocation strategy is employed to maximize computational efficiency while adhering to the optimal training hyperparameters. For each run, the optimal global batch size (GBZ) is calculated by $N$

756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809

| **Language** | **Original (B)** | **Sample** | **Sampled (B)** | **Weight(%)** |
|---|---|---|---|---|
| C | 52.33 | 1.00 | 52.33 | 5.84 |
| C++ | 67.43 | 1.00 | 67.43 | 7.53 |
| C# | 66.53 | 1.00 | 66.53 | 7.43 |
| Go | 12.76 | 1.00 | 12.76 | 1.43 |
| HTML | 260.27 | 0.05 | 13.01 | 1.45 |
| Java | 148.29 | 0.50 | 74.14 | 8.28 |
| JavaScript | 75.84 | 1.00 | 75.84 | 8.47 |
| Others | 307.22 | 1.00 | 307.22 | 34.31 |
| PHP | 75.67 | 1.00 | 75.67 | 8.45 |
| Python | 74.53 | 1.00 | 74.53 | 8.32 |
| Jupyter | 15.04 | 1.00 | 15.04 | 1.68 |
| Stack v2 | 55.46 | 1.00 | 55.46 | 6.19 |
| LeetCode | 5.56 | 1.00 | 5.56 | 0.62 |
| **Total** | **1,216.93** | - | **895.51** | **100.00** |

Table 3: Composition and sampling configuration of the training set by language.

and $D$ through a compute-aware heuristic. The maximum per-GPU micro batch size (MBZ), is then determined based on the model's size and training sequence length. This value can be slightly lower in multi-GPU setups. While using more GPUs can accelerate experiments, it often reduces Model FLOPs Utilization (MFU). We carefully manage this trade-off. The ideal number of GPUs, approximated by GBZ / MBZ, often requires adjustment due to hardware constraints such as integer divisibility. In making the adjustment, the priority is to maintain the GBZ as close to the optimal number as possible, given its critical impact on scaling law dynamics. Across our experimental runs, this strategy resulted in configurations using 8 to 128 GPUs, with a total compute equivalent to approximately 13,600 A100 GPU-days.

### A.4 Irreducible Loss Limit Derivation

A key theoretical divergence between the Chinchilla and Farseer scaling laws lies in their prediction of the model's performance limit. The Chinchilla formulation posits a non-zero irreducible loss, $E$, while the Farseer law's structure allows for a limit that can be zero. This difference is a direct consequence of their mathematical structures. For our fitted Chinchilla law (Equation 3), the limit is a substantial non-zero constant:

$$\lim_{N,D\to\infty}\left(0.2193+\frac{534.374}{N^{0.4853}}+\frac{76.0743}{D^{0.2983}}\right)=0.2193 \tag{3}$$

For our specific fitted Farseer law (Equation 4), while one term decays to zero, the second term converges to a very small, non-zero constant, making the final limit non-zero:

$$\begin{aligned}\lim_{N,D\to\infty} L(N,D) = &\underbrace{\lim_{N\to\infty} e^{-0.0047N^{0.239}-0.8188}}_{=0} \\ &+ \underbrace{\lim_{N,D\to\infty} e^{62.8936N^{-0.0614}-14.0414}\cdot D^{-e^{-0.0209N^{0.1943}-0.1826}}}_{=e^{-14.0414}} \\ \approx\ & 8.00\times 10^{-7}\end{aligned} \tag{4}$$

While our specific fit results in a near-zero irreducible loss rather than a true zero, it still stands in stark contrast to the much larger value predicted by the Chinchilla model. This motivates a deeper look into the theoretical argument for why such a limit should be possible, as it explores why, for models with sufficient context, the empirical entropy of a finite dataset can be expected to approach zero.

From an information-theoretic perspective, this discussion is centered on the entropy rate $h$ of the data, which can be expressed as the limit of the conditional entropy:

$$h=\lim_{n\to\infty} H(X_n|X_1,\ldots,X_{n-1})$$

The central question is whether $h$ must be a positive constant. While this may be true for an idealized, infinite data generating process, it is plausible to argue that the empirical entropy rate of any finite training corpus trends towards zero as the modeled context length $n$ becomes sufficiently large. To make this abstract concept more concrete, let us examine the practical realities of modern transformers.

Consider a model with a typical context window of $n = 4096$ tokens. Its objective is to predict the $4096$-th token given the prefix of $4095$ tokens. The space of all possible prefixes is vast, on the order of $V^{4095}$ where $V$ is the vocabulary size. Given the finite size of any real-world training corpus (e.g., trillions of tokens), this number is minuscule compared to the space of possible prefixes. Consequently, it is highly improbable that a specific, long prefix of 4095 tokens appears more than once in the entire dataset.

For the vast majority of training instances, the model is therefore presented with a prefix that is empirically unique. This uniqueness implies that the token that follows it is also unique from the dataset's point of view, making it deterministic in this context. In such cases, the empirical conditional probability for the next token is effectively 1 for a single outcome. The conditional entropy for such a unique prefix is therefore zero:

$$H(X_{4096}|X_1, \ldots, X_{4095}) = -\sum_x P(x|\text{prefix}) \log_2 P(x|\text{prefix}) \approx -(1 \cdot \log_2 1) = 0$$

A unique prefix leading to a deterministic next token is not a rare edge case but the dominant scenario when dealing with large context windows on finite data. This prevalence of empirically deterministic sequences suggests that as a model's context capacity increases, the average empirical entropy it is tasked with modeling should decrease. Therefore, a scaling law whose functional form permits a limit of or near zero, like Farseer, may provide a more accurate theoretical foundation for transformers with large context windows. Such models have the capacity to leverage long-range, near-deterministic patterns that are inherent to any finite data collection.

### A.5 The Use of Large Language Models (LLMs)

This article utilizes LLMs for polishing writing and helping to draw figures.