# SELF-PLAY PRETRAINING WITH ZERO DATA

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

## ABSTRACT

Advances in language modeling have been driven by scaling pretraining on ever more data. Yet, the training data is still largely curated on the model's behalf. A more general approach to pretraining would let the model learn to generate the data most useful for its own improvement. This would provide an effectively unbounded source of training data, limited by compute rather than human knowledge. We introduce **Self-Play Pretraining with Zero Data**, an initial proof-of-concept towards realizing this vision. Our procedure casts synthetic data generation as a search over the space of all computable structure, taking inspiration from Solomonoff induction. Starting from random initialization, two models learn in tandem: a generator proposes programs interpreted by a universal Turing machine, generating byte sequences, while a learner autoregressively predicts these byte sequences. The learner is trained with standard cross-entropy, while the generator is trained with reinforcement learning to produce sequences at the frontier of the learner's capabilities, yielding an adaptive curriculum. A universal Turing machine gives us a search space over all computable data-generating processes, imposing little domain-specific structure, and self-play searches over this space for useful training data. We test whether zero-shot performance on natural data improves predictably with self-play compute; this is a clean test of transfer since neither generator nor learner is trained on natural data. Across several natural datasets, zero-shot loss exhibits predictable scaling in compute. The models also exhibit in-context learning, and discover recognizable mathematical sequences during training.

## 1 INTRODUCTION

Advances in language modeling have been driven by pretraining on Internet data. Yet, the training data is still largely curated and constructed on the model's behalf through large-scale data curation efforts (Li et al., 2025; Soldaini et al., 2024; Penedo et al., 2023), data mixture design (Xie et al., 2023; Chen et al., 2026), and hand-designed synthetic generators (Gunasekar et al., 2023; Yang et al., 2025). A more generic approach to pretraining would let the model itself learn to generate training data that is most useful for its own improvement. Such an approach would extend the Bitter Lesson (Sutton, 2019) to the training data itself, minimizing hand-engineered inductive bias and creating a self-contained path to scaling, where compute alone can sustain continued improvement (Kim et al., 2026; Silver & Sutton).

As an initial step towards this vision, we introduce a self-play algorithm for pretraining from *zero data*. Starting from random initialization, two autoregressive transformers co-evolve: a generator proposes programs for a minimal *universal Turing machine* whose execution produces byte sequences, while a learner is trained on those sequences via next-token prediction. We use a universal Turing machine to make the space of all synthetic training data as expressive as possible while imposing minimal domain-specific structure: any computable data-generating process can, in principle, be represented as a program. However, the space of computable structure is enormous, and only a small subset of programs produce sequences that are useful for the learner. Moreover, whether a sequence is useful changes as the learner improves. This is why a self-play approach that adapts to the learner is natural: rather than specifying useful structure in advance, we let the generator discover which programs are most useful to the learner as training progresses. To encourage this behavior, the generator is trained with reinforcement learning using a learning-progress reward, shifting probability toward programs whose outputs lie near the frontier of the learner's current capabilities. These



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

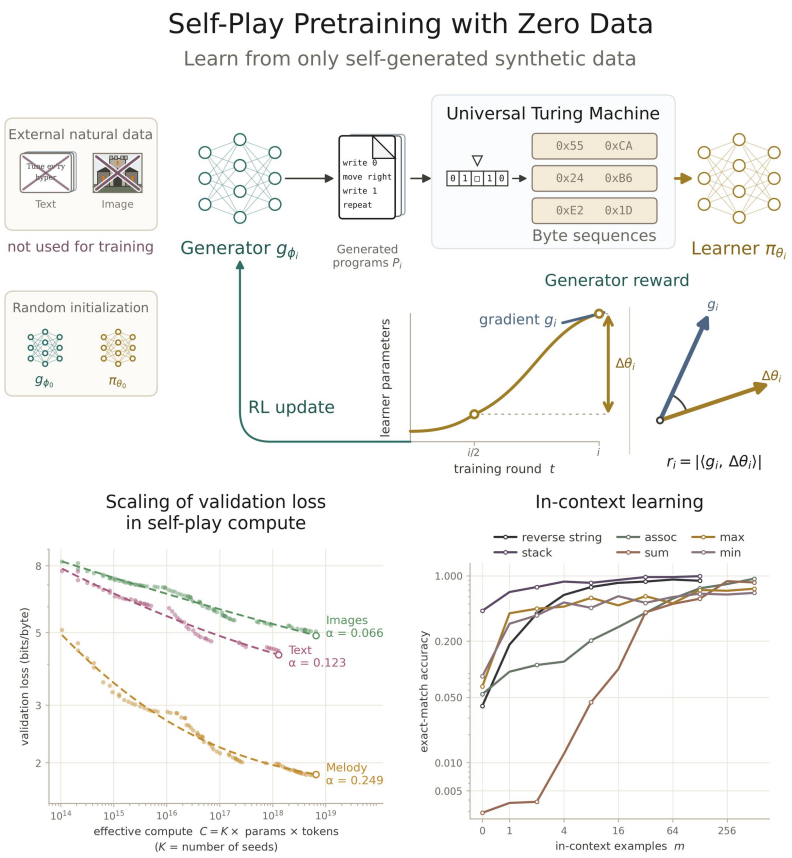



Figure 1: **Self-Play Pretraining with Zero Data**. Starting from randomly initialized models and using only synthetically generated data, self-play produces predictable scaling on held-out natural data. **(Top)** Our procedure casts synthetic data generation as search over the space of computable structure. A generator proposes programs, which are executed to produce byte sequences used to train a learner by next-token prediction. The generator is trained via reinforcement learning to propose programs near the frontier of the learner's capabilities, measured by how strongly the learner's gradients align with the learner's recent learning trajectory with respect to an AdamW-preconditioned inner product. **(Lower left)** Self-play produces predictable improvements in validation loss with compute across natural datasets. We show the compute-optimal frontier over model sizes, number of self-play rounds, and ensemble sizes. **(Lower right)** The resulting learner exhibits in-context learning on held-out tasks without any gradient updates. We report empirical success under greedy decoding as a function of the number of in-context examples $m$, averaged over independently sampled task instances.

design choices place our approach in the lineage of classical *universal prediction* (Solomonoff, 1964; Grau-Moya et al., 2024; Hutter, 2000; Bloem, 2025; Merhav & Feder, 1998), which formalizes how induction can be possible when the hypothesis class contains all computable data-generating processes and the learner has unbounded compute: we aim at an efficient computable approximation to universal prediction.

The resulting self-generated data is only valuable insofar as what the learner learns transfers to natural data. Our key hypothesis is that self-play over this space of computable data-generating processes can discover generic predictive regularities—such as copying, recursion, and hierarchical composition—that improve prediction on natural data. Importantly, we hypothesize that these regularities capture structure independent of *contingent information*—the particular facts, symbols, or modality of any one dataset—that can therefore transfer across data-generating processes. Indeed, prior work on formal, algorithmic, and non-linguistic pretraining distributions provides evidence that such cross-distribution transfer is possible (Grau-Moya et al., 2024; Papadimitriou & Jurafsky,

2020; Hu et al., 2025; Lee et al., 2026).

We test this hypothesis through a compute-optimal scaling law analysis. Concretely, we train randomly-initialized transformers at various scales via self-play and evaluate the resulting learners zero-shot on held-out datasets spanning natural language, images, speech, melodies, DNA, and mathematical sequences. For each dataset, we construct a compute-optimal frontier over model size, self-play rounds, and ensemble size. Across these diverse domains, a single family of self-play models exhibits predictable power-law improvements in zero-shot loss with compute, with scaling exponents comparable to those obtained by training directly on natural data. Importantly, this is a clean test of transfer since we deliberately run this process *tabula rasa*: both models are randomly initialized and all learner training data is generated through self-play. Thus, our experiments isolate the effect of our self-play procedure and test whether useful predictive structure can emerge *ex nihilo*. We also find that the learner develops in-context learning on completely held-out tasks, and the generator discovers known mathematical sequences.

## 2 Self-Play Pretraining with Zero Data

We introduce a self-play formulation of pretraining that involves searching over the space of computable structure. Every training sequence is the output of a program executed on a fixed universal Turing machine $U$ and these programs are produced by a learned generator. Our method has two components: (1) a **Learner** $\pi_\theta$, an autoregressive language model trained to predict program outputs, and (2) a **Generator** $g_\phi$, an autoregressive language model defined over programs for $U$. Both are transformers with the same architecture, trained from *random initialization*; the generator is trained via reinforcement learning (RL) and the learner is trained via next-token prediction. Importantly, we consider this *tabula rasa* setup to cleanly test whether self-play can generate structure that transfers to natural data.

Each round of self-play proceeds as follows:

1. **Program generation:** Sample $N$ programs from the generator $\{x_i\}_{i=1}^N \sim g_\phi$.
2. **Execution:** Run each program on $U$ to obtain output sequences $y_i = U(x_i, \omega_i)$, where $\omega_i$ is a random input tape.
3. **Learner and Generator update:** The learner takes one gradient step on the output sequences, optimizing the standard next-token loss. The generator takes a policy gradient step with a *learning-progress* reward that encourages the generator to propose programs at the frontier of the learner's capabilities. In addition, the generator is updated via a supervised fine-tuning objective on existing programs to mitigate catastrophic forgetting and on mutated programs to promote exploration.

### 2.1 Program space

We would like the generator's search space to be as expressive as possible while imposing little domain-specific structure. We therefore use programs for a minimal universal Turing machine as the substrate for generating synthetic data. Specifically, we use a Brainf*ck-like Turing-complete language, following Grau-Moya et al. (2024). Its primitive instructions manipulate a byte-valued tape, implement loops, and read or emit bytes. Because the language is universal, any computable data-generating process can, in principle, be represented as a program.

Let $x \in \mathcal{A}^{\leq L}$ denote a program generated over the machine's instruction alphabet. Executing $x$ on the universal machine $U$ with random input tape $\omega$ produces a byte sequence

$$y = U(x, \omega) \in \{0, \ldots, 255\}^T.$$

The random input tape allows a single program to represent a distribution over output sequences. These output bytes, rather than the programs themselves, constitute the learner's training data. We design the execution semantics so that every generated string is executable: programs cannot fail through syntax or memory errors, and execution always produces a bounded-length output. We defer the precise execution semantics and resource limits to Appendix E.

### 2.2 OBJECTIVES

**Program pool.** At each self-play round $e$, we construct a pool of programs

$$\mathcal{B}_e = \mathcal{B}_e^{\text{fresh}} \dot{\cup} \mathcal{B}_e^{\text{mut}} \dot{\cup} \mathcal{B}_e^{\text{replay}},$$

containing fresh samples from the current generator, local mutations of previously high-reward programs, and programs replayed from earlier rounds. Fresh samples provide global exploration, mutations refine promising regions of program space, and replay preserves useful structures discovered earlier in training. We write $M_e = |\mathcal{B}_e|$ for the number of programs in the pool. Details for mutation, replay, and the program bank are offered in Appendix G.

**Learner Update.** The learner is trained by standard next-token prediction on program outputs. For an output sequence $y$, define the per-sequence loss $\mathcal{L}(y;\theta)$ as the mean cross-entropy over its content tokens. If $y_i = U(x_i, \omega_i)$ is the output obtained by executing program $x_i$, the learner objective in round $e$ is

$$\mathcal{L}_{\text{learner}}(\theta; \mathcal{B}_e) = \frac{1}{M_e} \sum_{i \in \mathcal{B}_e} \mathcal{L}(y_i; \theta). \tag{1}$$

Thus fresh, mutated, and replay programs all train the learner.

**Generator Reward.** The generator's reward must be capable of identifying programs with useful structure from programs without any external feedback. Initially, we considered a reward based on how difficult a sequence is to predict, motivated by prior work on self-play (Dong & Ma, 2025; Bailey et al., 2026). However, this has a fundamental failure mode: a program can be made arbitrarily difficult without containing useful structure—for example, by injecting random bytes into an otherwise predictable sequence.

To avoid this degeneracy, we evaluate a new program based on whether it builds on what the learner has actually been able to learn. Intuitively, the learner's change in parameters summarizes this: learning signals arising from reusable structure accumulate, whereas we expect that idiosyncratic effects that are not learnable do not. Concretely, we reward the generator for producing programs whose learner gradients align with the learner's current learning trajectory. Let

$$p(e) = \lfloor e/2 \rfloor, \qquad \delta\theta_e = \theta_{p(e)} - \theta_e,$$

where $\theta_{p(e)}$ is the learner checkpoint at the lookback horizon. The reward for program $x_i$ with output $y_i$ is

$$r_i = |\, \langle \nabla_\theta \mathcal{L}(y_i; \theta_e), P_e \odot \delta\theta_e \rangle \,|, \tag{2}$$

where

$$P_e = \frac{\text{lr}}{\sqrt{\hat{v}_e} + \epsilon}$$

is the diagonal AdamW step operator obtained from the learner's optimizer state. We use the lookback window of $\lceil e/2 \rceil$ so that signals which take a long time to appear can be measured. The growing window helps average over short-term fluctuations and produce a signal which becomes more stable as training progresses, but allows early mistakes to eventually be forgotten.

We provide some additional intuition below, but we emphasize that we selected this reward after searching through several possibilities at small scale. Detailed analysis can be found in table 4. Formally, this reward is a preconditioned gradient-alignment score, between the learner's gradient on a program's output and the learner's parameter movement over the lookback window, with the diagonal AdamW preconditioner defining the inner product; we found that using this pre-conditioning was important in line with Thrush et al. (2026). Intuitively, this reward favors programs that are not yet mastered, but whose structure extends what the learner has already shown it can learn. We expect that programs that are already mastered induce nearly zero gradients, while programs containing unrelated or unlearnable structure induce gradients that do not align with the learner's parameter movement. Both receive little reward. Instead, high reward is assigned to programs that induce substantial learning, but only in directions congruent with the learner's recent progress; this concentrates the generator on the frontier of the learner's current capabilities. We avoid materializing full gradients by using forward mode automatic differentiation (Griewank & Walther, 2008) to calculate Equation 2.[1]

[1]Forward mode kernel implemented in `https://github.com/amorehead/jvp_flash_attention`

**Policy-Gradient RL Objective.** We train the generator using a KL-regularized expected reward

$$J_{\mathrm{RL}}(\phi) = \mathbb{E}_{x\sim g_\phi}[r(x)] - \beta\,\mathrm{KL}(g_\phi \| g_0), \tag{3}$$

where $\beta$ is the KL regularization coefficient and $g_0$ is the fixed uniform program prior defined as

$$g_0(x) = |\mathcal{A}|^{-\ell(x)}.$$

Here $\ell(x)$ is the number of tokens up to and including the terminating `F`. This is the natural analog of the Solomonoff prior $2^{-|p|}$ (Solomonoff, 1964), which weights programs according to description length. The generator is initialized near $g_0$ and regularized toward it throughout training.

Since the vanilla policy gradient estimator is high variance, we consider a GRPO (batch-level) based estimator (Guo et al., 2025). Let $\bar{r}_e$ and $\sigma_{r,e}$ denote the mean and standard deviation, respectively, of the rewards $\{r_i\}_{i\in\mathcal{B}_e}$ over the full round pool. We define the advantage of program $i$ as

$$A_i = \frac{r_i - \bar{r}_e}{\sigma_{r,e} + \epsilon} - \beta\left(\log g_\phi(x_i) - \log g_0(x_i)\right).$$

Since our bank consists of off-policy samples, we use a sequence-level importance ratio correction $\rho_i = \frac{g_\phi(x_i)}{g_{\phi_{\mathrm{old},i}}(x_i)}$ (Zheng et al., 2025). It is equal to one for fresh on-policy samples; for replay samples, its denominator is the sampling probability stored when the program originally entered the replay bank. The policy-gradient term is then

$$\mathcal{L}_{\mathrm{PG}}(\phi) = -\frac{1}{|\mathcal{B}_e \setminus \mathcal{B}_e^{\mathrm{mut}}|} \sum_{i\in\mathcal{B}_e\setminus\mathcal{B}^{\mathrm{mut}}} \mathrm{stopgrad}(\rho_i)\,\log g_\phi(x_i)\,\mathrm{stopgrad}(A_i). \tag{4}$$

Mutation rows are excluded because they were not sampled from a proposal distribution with a well-defined log probability. We clip the sequence-level importance ratio to ensure $\rho_i \in [e^{-20}, e^{20}]$.

**Expert Iteration.** To prevent forgetting, we replay previous programs by distilling high-reward programs back into the generator using reward-weighted supervised fine-tuning over the full pool $\mathcal{B}_e$. This is a technique used to mitigate forgetting in pretraining and RL (Ibrahim et al., 2024; Schaul et al., 2016). We assign each program a normalized sequence-level weight

$$w_i = \frac{[r_i]_+}{\sum_{j\in\mathcal{B}_e}[r_j]_+}, \qquad [r]_+ \equiv \max(r, 0),$$

and optimize

$$\mathcal{L}_{\mathrm{EI}}(\phi; \mathcal{B}_e) = -\sum_{i\in\mathcal{B}_e} w_i \log g_\phi(x_i).$$

The generator's full training objective is

$$\mathcal{L}_{\mathrm{generator}}(\phi) = \mathcal{L}_{\mathrm{PG}}(\phi) + \lambda_{\mathrm{EI}}\,\mathcal{L}_{\mathrm{EI}}(\phi; \mathcal{B}_e), \tag{5}$$

where $\lambda_{\mathrm{EI}} = 1.0$ controls the strength of the reward-weighted SFT term.

### 2.3 Architecture and Tokenization

The learner and generator are independently parameterized decoder-only Llama transformers with identical architecture (Touvron et al., 2023). We use byte-level tokenization with a fixed vocabulary of 256 byte values because the universal machine produces raw bytes, and because it enables a clean, modality-agnostic evaluation of universal prediction: how well the model predicts the next byte in arbitrary sequences encoding text, images, audio, or other data. Programs and outputs are prefixed by the bytes `S` and `O`, respectively. During program generation, logits are restricted to the eight Brainf*ck instructions, ten canonical single-byte macro instructions, and the end-of-program token `F`, whereas output sequences may contain any byte value.

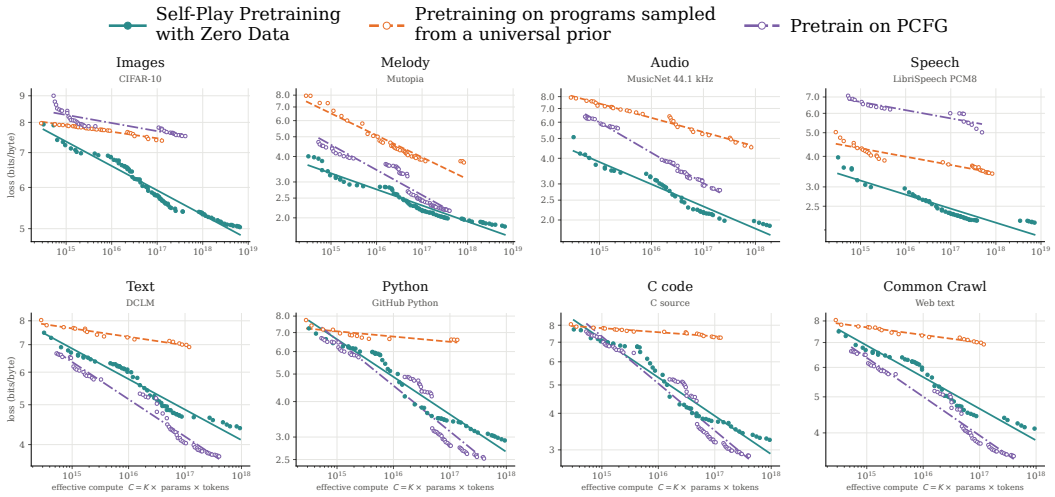



Figure 2: **Self-play learns predictive structure that transfers across modalities.** We compare self-play to two fixed synthetic pretraining distributions: programs sampled from a universal prior over Brainf*ck programs and probabilistic context-free grammars (PCFGs). Sampling from the universal prior scales substantially more slowly than self-play, showing that access to a universal space of programs alone is insufficient without the adaptive curriculum. Pretraining on PCFG transfers strongly to language-like domains, but its benefits are less consistent across non-language modalities. In contrast, self-play exhibits predictable scaling across images, melody, audio, speech, text, and code despite using minimal inductive bias. The compute-optimal frontier ends where we do not find further models with smaller loss than the largest-compute model shown.

# 3 Empirical Results

## 3.1 Universal zero-shot transfer scaling laws

In standard pretraining, scaling compute typically entails both increasing model size and training the model on more natural data. Our scaling experiments study whether increasing compute via self-play, without any natural data, produces predictable improvements in zero-shot performance on held-out natural data.

**Scaling recipe.** We follow the scaling methodology of Kim et al. (2026); Wen et al. (2026). At each model scale, we tune hyperparameters to *local optimality*, in the sense defined in Kim et al. (2026), via coordinate descent on a geometrically-spaced grid. Because our training objective is entirely synthetic, it does not provide an obvious criterion for selecting hyperparameters; poorly tuned hyperparameters could prevent clean scaling laws. We therefore use validation loss averaged across DCLM and DNA as a model-selection signal. This introduces limited leakage through hyperparameter selection, but natural data are never used for gradient updates.

Because tuning every possible hyperparameter at every scale is infeasible, we restrict the search to the most important hyperparameters based on preliminary experiments: the learner learning rate, the generator-to-learner learning-rate *ratio*, the batch size, and the generator KL regularization coefficient $\beta$. Due to compute constraints, we fix a large maximum training budget of 34.36B tokens rather than separately tuning the number of self-play rounds. Because we use only a short fixed warmup followed by a constant learning rate, every intermediate checkpoint is equivalent to a run stopped at that token budget. Thus, a single long run allows us to optimize over training duration retrospectively when constructing the scaling laws. When tuning batch size, we hold the total number of training tokens fixed; when the batch size changes, the number of rounds is adjusted accordingly to preserve the total token budget. Across all scales, we use a context length of 4096 tokens. We find that ensembling across randomly initialized models is useful. Therefore, at the locally-optimal hyperparameters, we train $K$ independent seeds per scale and form ensembles averaging the models' predictive distributions.

For each dataset and algorithm, we construct a *compute-optimal* frontier (Kaplan et al., 2020) over model sizes, checkpoints, and ensemble sizes. A point is on the frontier if and only if it achieves lower validation loss than every observed configuration with less than or equal compute.

We fit each compute-optimal frontier with the asymptotic power law

$$L(C) = E + AC^{-\alpha},$$

where $L$ is the observed loss in bits per byte and $E$ is a fitted asymptotic loss floor. We fit the model independently for each dataset.

**Self-play exhibits universal zero-shot power-law scaling in compute.** As shown in Figure 1, we observe scaling laws over a diverse range of modalities: text, images, and music. We present additional results in Figure 5. The scaling laws over these modalities are broadly similar, as seen in Table 1, (with DNA as the exceptional case). We discuss the implications of this in Section 4. In brief, we expect this to be the case when learning universal structure rather than contingent knowledge is the bottleneck to scaling. Importantly, these scaling results are entirely zero-shot: that is, they arise without any gradient steps on any of the evaluation datasets.

**Pretraining on a fixed universal program prior exhibits slow scaling.** To isolate the value of self-play, we compare self-play against a non-adaptive baseline over exactly the same program space; we use the same mixture of validation loss on DCLM and DNA. Instead of learning a distribution over programs, the baseline samples programs from a fixed Solomonoff-style prior (Solomonoff, 1964): instruction tokens are drawn i.i.d. until termination, giving full support to every finite program while favoring shorter descriptions. Thus, both methods have access to the same universal space of computable structure; they differ only in whether the sampling distribution adapts to the learner. Figure 2 shows that fixed sampling scales substantially more slowly, demonstrating that access to a universal program space alone is not enough—self-play must learn where in that space to allocate training compute. We offer additional evaluation datasets in Figure 5.

**PCFG pretraining is effective on language-like domains but lacks broad cross-domain transfer.** In Figure 2, we also compare against pretraining on probabilistic context-free grammars (PCFGs), which provide a hand-designed source of hierarchical and compositional structure particularly well suited for language; for details of PCFG data generation see Appendix H. We expect pretraining on PCFG to be highly competitive on language-like domains, but its inductive bias is specialized to a particular class of structure. In contrast, our self-play procedure is, in principle, universal. Consistent with this interpretation, PCFG pretraining is stronger on text and code, where its inductive bias is well matched, while self-play substantially outperforms it on images, music, audio, and speech. Thus, self-play does not always match the performance of a specialized prior on domains where that prior is particularly well suited; however, it learns structure that transfers more broadly across modalities. We find that models trained on PCFG and the universal prior fail on our ICL evaluations in Figure 3.

### 3.2 In Context Learning

An emergent property of large language models is their ability to infer a task from examples provided in context and apply the inferred rule to new inputs (Brown et al., 2020). Beyond measuring held-out loss, in-context learning provides a complementary test of whether self-play pretraining has produced a model that can infer latent structure in sequences. We evaluate our learner's performance on several in-context learning tasks in Figure 1. We plot the empirical success rate under greedy $\arg\max$ decoding as a function of the number of in-context examples, $m$. Importantly, the learner does not receive any additional gradient updates or fine-tuning. Rather, the learner must infer the latent structure underlying the sequence entirely in context. From the perspective of universal prediction and Solomonoff induction, this is precisely the kind of behavior we would hope to emerge after our self-play procedure. Section D contains task-specific details.

**Self-play induces broad ICL where fixed synthetic pretraining does not.** In Figure 3, we see that the model can achieve almost 100% accuracy on REVERSE STRING (Delétang et al., 2023), STACK (Delétang et al., 2023), and ASSOCIATIVE RECALL (Ba et al., 2016; Graves et al., 2014) tasks after a sufficient number of ICL examples. The ASSOCIATIVE RECALL task shows that our

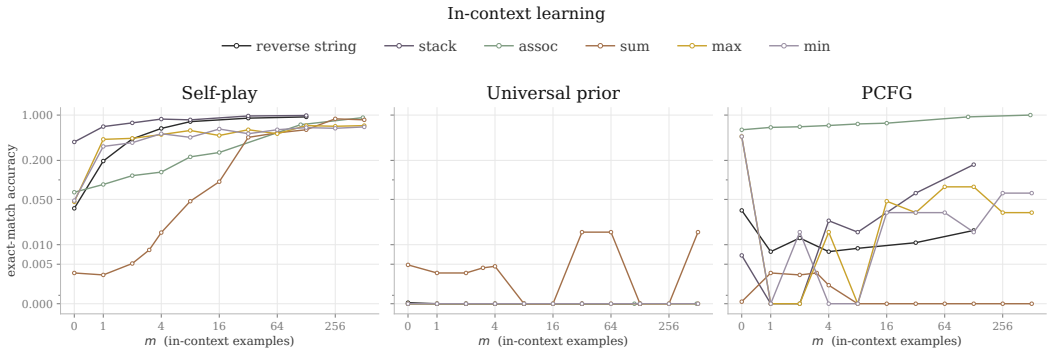



Figure 3: **Comparing ICL behavior across methods.** Self-play pretraining yields consistent improvements in ICL performance across all six tasks. In contrast, universal prior pretraining shows little evidence of effective ICL, while PCFG pretraining performs strongly on associative recall but transfers only weakly to the remaining tasks.

model is capable of contextual search, the REVERSE STRING task shows that our model is capable of dynamically indexing, and STACK shows that our model is capable of learning to simulate a context-free grammar. In addition, we show that our model is capable of learning standard mathematical relations in-context (MAX, MIN, SUM). This is a natural test given that our model has been trained on short programs which could naturally express the application of mathematical functions. Before any examples have been shown ($m = 0$) the model already has a 6%-8% prediction accuracy on the MAX and MIN tasks due to its prior on copying previously produced tokens. In contrast, the models trained on PCFG and on samples from the universal prior over programs cannot learn all of these ICL tasks (Figure 3).

## 4 Explaining Self-Play Scaling laws via Universal Data Ansatz

Our experiments show that (1) prediction on natural data improves predictably even though the learner does not train on any natural data and (2) the observed scaling exponents are comparable to those obtained from standard pretraining on particular domains. We propose a simple interpretation of these two results by separating two sources of predictive information that are ordinarily entangled in natural data: *contingent information*, which is specific to the particular world or distribution that generated the data, and *universal predictive structure*, which is shared across many data-generating processes.

**Decomposing natural-data scaling.** Hoffmann et al. (2022) model loss as a function of model size $N$ and natural dataset size $D$ using the ansatz

$$L = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}. \tag{6}$$

We refine the data term by treating contingent information and universal structure as separate resources:

$$L = E + \frac{A}{N^{\alpha}} + \frac{B}{D_c^{\beta}} + \frac{C}{D_u^{\gamma}}, \tag{7}$$

where $D_c$ denotes the effective amount of contingent information available to the model and $D_u$ the effective amount of universal predictive structure.

**Explaining self-play power law scaling in compute.** We first use the ansatz in Equation 7 to understand why we might see power law scaling in self-play compute. Since our models are not trained on natural data, contingent information is fixed as training proceeds. We absorb the third term in Equation 7 into a dataset-specific constant $E'$. Only universal predictive structure generated through self-play can grow, giving

$$L = E' + \frac{A}{N^{\alpha}} + \frac{C}{(D_u(T))^{\gamma}}. \tag{8}$$

Here we write $D_u(T)$ to indicate the amount of universal data generated via self-play at time $T$. If self-play generates universal structure at a power-law rate,

$$D_u(T) \propto T^{\eta}, \tag{9}$$

then

$$L = E' + \frac{A}{N^{\alpha}} + \frac{C'}{T^{\gamma\eta}}. \tag{10}$$

Thus, the ansatz directly predicts the form of the scaling law we observe: natural-data loss can improve as a power law in the amount of self-play training data $T$, despite no natural data being added during training, at the cost of a larger irreducible error[2].

**Comparing self-play and natural-data exponents.** Now, we show that, with some additional assumptions, the conventional one-term data-scaling law conflates gains from learning contingent information with gains from learning universal structure. In standard pretraining, increasing the amount of natural data $D$ increases both resources. Suppose

$$D_c(D) \propto D^{\nu}, \qquad D_u(D) \propto D^{\mu}. \tag{11}$$

Substituting into Equation (7) gives

$$L = E + \frac{A}{N^{\alpha}} + \frac{B'}{D^{\beta\nu}} + \frac{C'}{D^{\gamma\mu}}. \tag{12}$$

Asymptotically, the more slowly decaying of the two terms dominates, so we expect that a single fitted data exponent recovers

$$\beta_{\text{obs}} \approx \min\{\beta\nu, \gamma\mu\}. \tag{13}$$

We observe self-play exponents, which estimate $\gamma\eta$, comparable to those reported for direct training on the corresponding natural modalities (table 1). This comparison is informative because self-play, in our setting, can improve only by increasing universal predictive structure, whereas natural data pretraining can improve through both universal and contingent information. We cautiously interpret this as suggesting that learning universal structure may account for an important component of the improvements obtained by scaling natural data.

## 5 Conclusion and Discussion

Our results show that predictable improvements in prediction on natural data can emerge even when no natural data is used for training. The similarity between our self-play scaling exponents and those observed under direct natural-data pretraining suggests that learning universal structure may account for an important component of conventional gains. At the same time, self-play cannot recover contingent facts about the world, so we view our approach as complementary to, rather than a replacement for, natural data. We present preliminary results in this direction in Section A.1. More broadly, our results raise the exciting possibility that self-generated experience could continue expanding the learning frontier when natural data is exhausted. While our results are promising, they are small scale ($< 25$ M parameters); this reflects compute limitations and our approach, in principle, has no obvious algorithmic bottlenecks preventing further scaling. Whether this behavior persists empirically for larger models remains an important direction for future work. Moreover, although the scaling laws are encouraging, they are not as clean as conventional pretraining scaling laws; we suspect this is because RL introduces additional noise into training that is not present in standard pretraining. future work. We defer a broader discussion to Appendix.

[2]This must be the case for finite context lengths, but is not necessarily the case when the model has an infinite context length which it could theoretically use to learn any domain in-context.

## AI USE STATEMENT

In this work, we used generative AI tools for help develop theoretical models or conceptual frameworks, coding assistance, implement methods, writing assistance. We have not used generative AI tools for generating synthetic data sets, cleaning and reformatting datasets, proposing or refining hypotheses, assisting with translation, designing or providing feedback on research methodology or experiments, or generating research ideas, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for creating and modifying scientific figures and creation of artifacts, supporting qualitative and thematic data analysis, and interpreting results. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES


Armen Aghajanyan, Lili Yu, Alexis Conneau, Wei-Ning Hsu, Karen Hambardzumyan, Susan Zhang, Stephen Roller, Naman Goyal, Omer Levy, and Luke Zettlemoyer. Scaling laws for generative mixed-modal language models. In *International Conference on Machine Learning*, pp. 265–279. PMLR, 2023.

AITDCC. Algorithmic information theory data compression challenge official data archive. `https://github.com/AITDCC/aitdcc.github.io/tree/16887f9d68f273503eddb69e0c29aa7b4f39ba7a`. `data/data.zip`, member `data/B`, GitHub repository, commit `16887f9d68f273503eddb69e0c29aa7b4f39ba7a`.

Jimmy Ba, Geoffrey Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pp. 4338–4346, Red Hook, NY, USA, 2016. Curran Associates Inc. ISBN 9781510838819.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance, 2026. URL `https://arxiv.org/abs/2604.20209`.

Peter Bloem. Universal pre-training by iterated random computation, 2025. URL `https://arxiv.org/abs/2506.20057`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mayee F. Chen, Tyler Murray, David Heineman, Matt Jordan, Hannaneh Hajishirzi, Christopher Ré, Luca Soldaini, and Kyle Lo. Olmix: A framework for data mixing throughout lm development, 2026. URL `https://arxiv.org/abs/2602.12237`.

Mutopia Project contributors. The mutopia project. URL `https://www.mutopiaproject.org/`.

Santiago Cuervo and Ricard Marxer. Scaling properties of speech language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 351–361, 2024.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy, 2023. URL `https://arxiv.org/abs/2207.02098`.

Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14114–14136. PMLR, 13–19 Jul 2025. URL `https://proceedings.mlr.press/v267/dong25h.html`.

Marc Finzi, Shikai Qiu, Yiding Jiang, Pavel Izmailov, J. Zico Kolter, and Andrew Gordon Wilson. From entropy to epiplexity: Rethinking information for computationally bounded intelligence, 2026. URL https://arxiv.org/abs/2601.03220.

Jordi Grau-Moya, Tim Genewein, Marcus Hutter, Laurent Orseau, Grégoire Delétang, Elliot Catt, Anian Ruoss, Li Kevin Wenliang, Christopher Mattern, Matthew Aitchison, and Joel Veness. Learning universal predictors, 2024. URL https://arxiv.org/abs/2401.14953.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. 2014. URL https://arxiv.org/abs/1410.5401.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008. ISBN 9780898717761. URL https://books.google.com/books?id=xoiiLaRxcbEC.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023. URL https://arxiv.org/abs/2306.11644.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling laws for autoregressive generative modeling, 2020. URL https://arxiv.org/abs/2010.14701.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hen-

nigan, 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, 2022. URL https://arxiv.org/abs/2203.15556.

Michael Y Hu, Jackson Petty, Chuan Shi, William Merrill, and Tal Linzen. Between circuits and chomsky: Pre-pretraining on formal languages imparts linguistic biases. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9691–9709, 2025.

Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity, 2000. URL https://arxiv.org/abs/cs/0004001.

Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L. Richter, Quentin Anthony, Timothée Lesort, Eugene Belilovsky, and Irina Rish. Simple and scalable strategies to continually pre-train large language models, 2024. URL https://arxiv.org/abs/2403.08763.

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 preprint arXiv:2001.08361*, 2020.

Konwoo Kim, Suhas Kotha, Percy Liang, and Tatsunori Hashimoto. Pre-training under infinite compute. In *International Conference on Learning Representations*, volume 2026, pp. 74596–74636, 2026.

Dan Lee, Seungwook Han, Akarsh Kumar, and Pulkit Agrawal. Training language models via neural cellular automata, 2026. URL https://arxiv.org/abs/2603.10055.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models, 2025. URL https://arxiv.org/abs/2406.11794.

N. Merhav and M. Feder. Universal prediction. *IEEE Transactions on Information Theory*, 44(6): 2124–2147, 1998. doi: 10.1109/18.720534.

Metamath contributors. set.mm: Metamath proof database. https://github.com/metamath/set.mm/tree/bcfef9892b6103ba9046bf683b4903d2ad081a41. GitHub repository, commit bcfef9892b6103ba9046bf683b4903d2ad081a41.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

NCBI. Homo sapiens genome assembly GRCh38. NCBI Datasets. URL https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.26/. RefSeq assembly accession GCF_000001405.26.

Isabel Papadimitriou and Dan Jurafsky. Learning music helps you read: Using transfer to study linguistic structure in language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6829–6839, 2020.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: Outperforming curated corpora with web data, and web data only, 2023. URL https://arxiv.org/abs/2306.01116.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016. URL https://arxiv.org/abs/1511.05952.

Arnav Shah, Junzhe Li, Parsa Idehpour, Adibvafa Fallahpour, Brandon Wang, Sukjun Hwang, Bo Wang, Patrick D. Hsu, Hani Goodarzi, and Albert Gu. dnahnet: A scalable and hierarchical foundation model for genomic sequence learning, 2026. URL https://arxiv.org/abs/2602.10603.

David Silver and Richard Sutton. Welcome to the era of experience.

Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Pete Walsh, Luke Zettlemoyer, Noah A. Smith, Hannaneh Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. Dolma: an open corpus of three trillion tokens for language model pretraining research, 2024. URL https://arxiv.org/abs/2402.00159.

Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1): 1–22, 1964.

Richard S. Sutton. The bitter lesson. Incomplete Ideas (blog), 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Tristan Thrush, Sung Min Park, Herman Brunborg, Luke Bailey, Marcel Rød, Neil Band, Christopher Potts, and Tatsunori Hashimoto. Synthetic data for any differentiable target. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=Iudff0keb0.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Kaiyue Wen, David Leo Wright Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=2J51qUZ0iG.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V. Le, Tengyu Ma, and Adams Wei Yu. Doremi: Optimizing data mixtures speeds up language model pretraining, 2023. URL https://arxiv.org/abs/2305.10429.

Zitong Yang, Neil Band, Shuangping Li, Emmanuel Candes, and Tatsunori Hashimoto. Synthetic continued pretraining. In *International Conference on Learning Representations*, volume 2025, pp. 44379–44421, 2025.

Ori Yoran, Kunhao Zheng, Fabian Gloeckle, Jonas Gehring, Gabriel Synnaeve, and Taco Cohen. The kolmogorov test: compression by code generation. In *International Conference on Learning Representations*, volume 2025, pp. 87896–87926, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025. URL https://arxiv.org/abs/2507.18071.

# A Additional experimental results

## A.1 Pre-Pretraining with Self-Play Accelerates Pretraining on Natural Data

Our scaling results show that self-play produces transferable structure; a natural follow-up question is whether that structure remains useful once natural data becomes available. We therefore treat self-

| Modality / dataset | Our $b$ | Literature $b$ | Ref. |
|---|---|---|---|
| *Text* | | | |
| text (dclm) | 0.123 | 0.048–0.099$^\dagger$ | Henighan et al. (2020); Aghajanyan et al. (2023) |
| *Images* | | | |
| CIFAR-10 (HWC interl.) | 0.066 | | |
| CIFAR-10 image bytes | 0.145 | 0.065$^\dagger$–0.10 | Aghajanyan et al. (2023); Henighan et al. (2020) |
| *Audio / speech* | | | |
| audio 16-bit PCM | 0.141 | | |
| audio 8-bit PCM | 0.260 | 0.12$^\dagger$–0.14$^\dagger$ | Cuervo & Marxer (2024); Aghajanyan et al. (2023) |
| MIDI (Mutopia, 16th note grid) | 0.249 | – | |
| *Math / formal* | | | |
| Metamath set.mm | 0.129 | 0.17 | Henighan et al. (2020) |
| *Biological sequences* | | | |
| DNA (8-symbol) | 0.435 | 0.01–0.06 | Shah et al. (2026) |
| *Code* | | | |
| AITDCC C source | 0.116 | 0.17$^\dagger$ | Aghajanyan et al. (2023) |
| Python source (GitHub) | 0.113 | | |

Table 1: Per-modality compute exponents $b$ from fits $L(C) = A\,C^{-b} + E$. Values marked $^\dagger$ are derived from published Chinchilla-form fits via $b = \alpha\beta/(\alpha+\beta)$ under the compute-optimal allocation. Where a range is given, values correspond to the citations in order. Dashes indicate that the authors could not find a published exponent for that modality. Exponents from SP are broadly similar to exponents from pre-training on the literature, if a bit higher. A detailed description and an illustration of the datasets can be found in appendix B

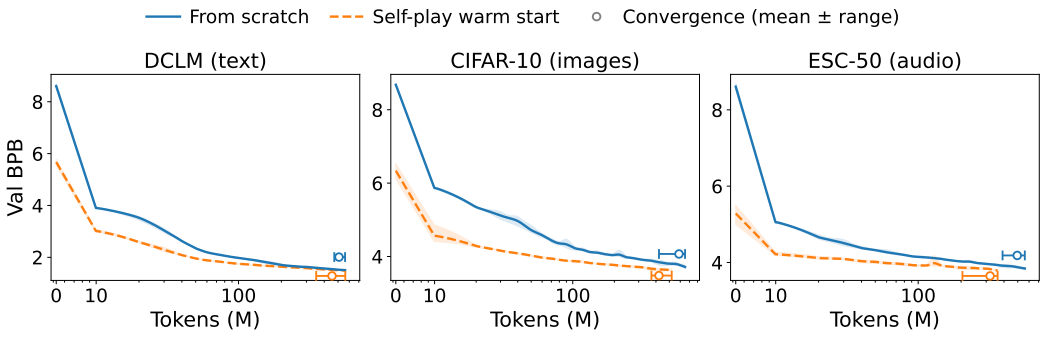



Figure 4: **Self-play pre-pretraining accelerates pretraining on natural data.** Validation BPB during pretraining of a 24.4M model on three byte-encoded modalities, from random initialization (blue) and from the final self-play checkpoint (orange), at each arm's best hyperparameters (mean over 4 seeds; shaded band spans the seed range). The leftmost point (marked 0) is the validation loss before any natural-data training — for the warm start, its zero-shot transfer — and the token axis is logarithmic from the first evaluation onward. Open markers show the mean tokens consumed at convergence, with horizontal bars spanning the seed range (markers vertically offset for legibility); Throughout training, the warm start reaches every loss level first.

play as *pre-pretraining* (Hu et al., 2025): a learner produced by self-play is used as the initialization for ordinary pretraining on natural data, and we compare it against the same architecture pretrained from random initialization.

**Setup.** We compare downstream pretraining of a 24.4M-parameter model from two initializations: random weights ("from scratch") and the final self-play checkpoint ("self-play warm start"). We evaluate both on DCLM text, CIFAR-10 images, and ESC-50 audio, using the same fixed natural-data corpus for each modality. Both methods are trained to convergence, with learning rate and weight decay tuned separately for each.

**Results** Figure 4 shows that self-play pre-pretraining accelerates downstream training across all three modalities. The warm-start models start with a lower loss than random initialization, as expected, and retain an advantage throughout training, reaching a low validation-loss level with fewer natural-data tokens. The savings are substantial on ESC-50 (320M vs. 496M tokens) and CIFAR-10 (421M vs. 588M). By the end of our training protocol, the loss gap has narrowed considerably, indicating that the clearest benefit of self-play pre-pretraining is accelerated learning from natural data.

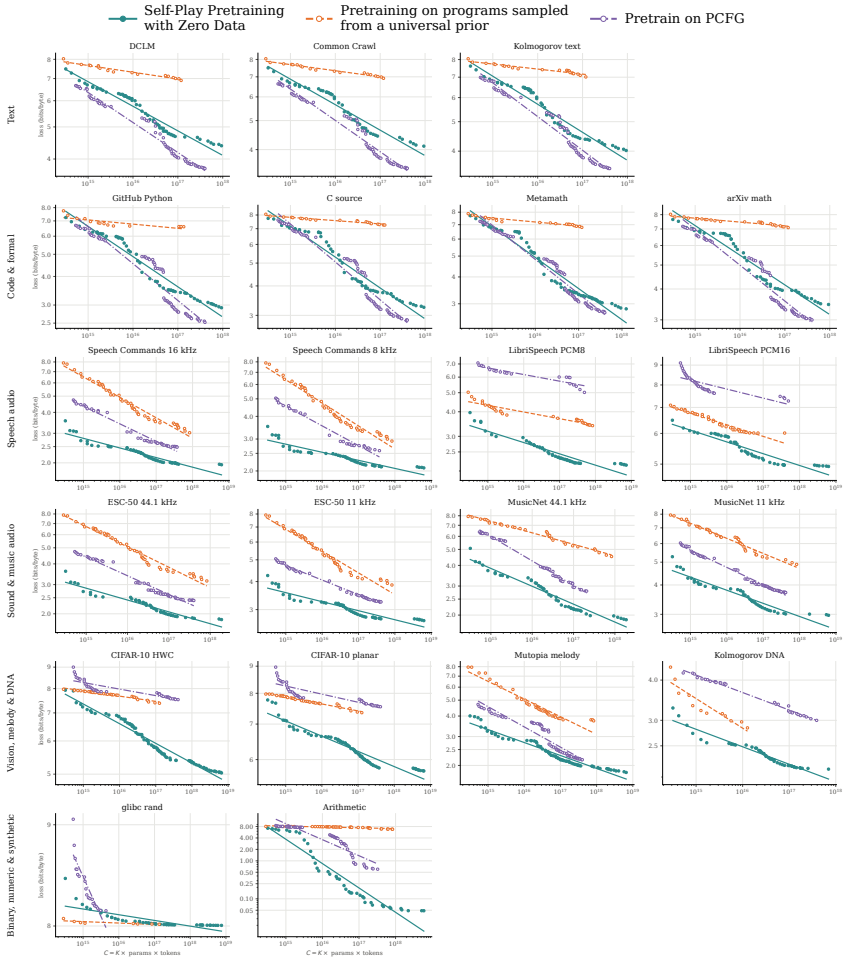



Figure 5: **Scaling laws across all evaluation datasets.**

## A.2 Analyzing the SUM task

## A.3 Pre-pretraining details

After the validation split, the training corpora contain approximately 255M (DCLM), 146M (CIFAR-10), and 152M (ESC-50) tokens, and runs repeat this data over epochs, terminating at approximate convergence. The learning rate is held constant until validation BPB plateaus (improvement $< 0.005$ for 5 consecutive evaluations), then decayed to zero over a 200-step cosine schedule; we report the *converged BPB*, the validation loss after this final decay. Learning rate and weight decay are tuned separately for each arm, from LR $\in \{10^{-3}, 3 \times 10^{-3}, 10^{-2}\}$ and WD $\in \{0.0, 0.1, 0.3, 0.8\}$ with 4 seeds per configuration; per arm we select the configuration with the lowest mean converged BPB across seeds. We do not count the compute spent on self-play pre-pretraining itself in this comparison: it is a one-time cost that amortizes across downstream training runs — here, a single self-play checkpoint initializes the warm arm on all three modalities — in the

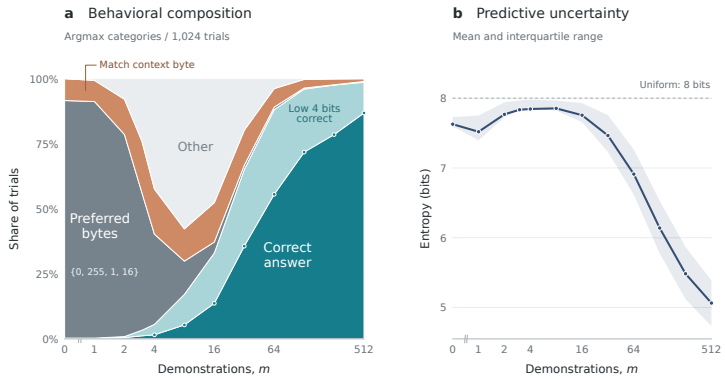



Figure 6: **Interpretable ICL behavior on SUM task.** **(a)** Distribution over the model's inferred strategies as the number of ICL examples increases. **(b)** Predictive entropy over the same trajectory, showing an initial loss of confidence followed by increasing certainty as the correct strategy emerges.

same way that a pretrained checkpoint is reused across many fine-tuning tasks.

## B BENCHMARK DETAILS

Here we collect more details on the benchmarks. To test the ability of the model to predict intrinsically different kinds of "natural" sequences, we designed a diverse set of benchmarks generated by different processes. Regardless of the provenance, they share the same interface, obtained by encoding them as byte sequences.

### B.1 NATURAL TEXT

To construct this benchmark, we used DCLM-Baseline-1.0, a filtered collection of web text extracted from Common Crawl. Explicitly, we sampled the ten global DCLM shards so that the benchmark did not come from only one part of the collection. Each DCLM record is stored in a JSON container, but the benchmark retains only its text field. We encode this text directly as UTF-8 bytes, concatenate the document texts within each shard, and divide the result into fixed windows for next-byte prediction. The predictor therefore has to exploit regularities such as spelling, punctuation, word structure, and local syntax through the same byte interface used for every other benchmark.

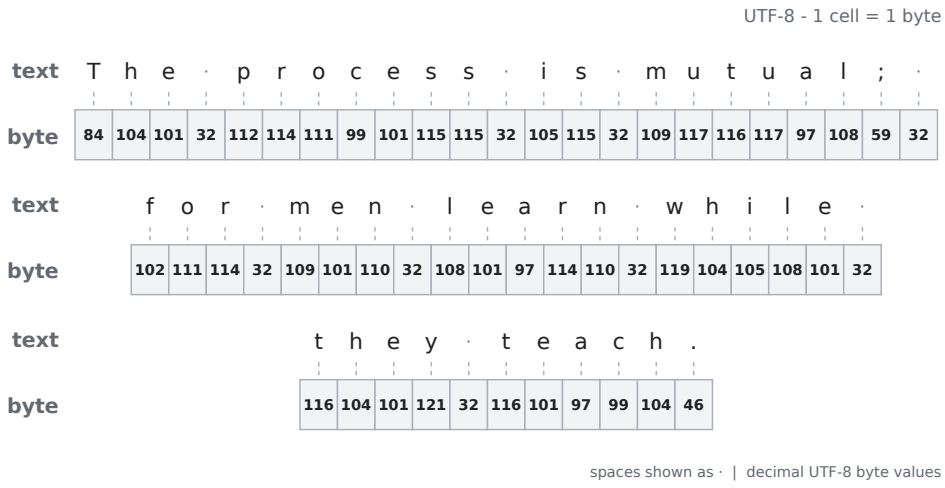



Figure 7: Illustration of the natural-text encoding for an illustrative ASCII excerpt. Each cell shows one decimal UTF-8 byte; middle dots mark spaces (byte 32).

### B.2 NATURAL IMAGES

To construct this benchmark, we used the official CIFAR-10 test batch. It contains $32 \times 32$ color images, each paired with one of ten class labels. Since our task is next-byte prediction rather than classification, we remove the labels and retain only the unsigned 8-bit RGB pixel values. We also exclude filenames, image headers, compression, and other metadata, so the predictor sees the common byte interface.

A two-dimensional image must still be arranged as a one-dimensional byte sequence. We included two lossless encodings of exactly the same images. The planar (CHW) encoding traverses each image row by row, storing all red values, then all green, and then all blue. The interleaved (HWC) encoding uses the same pixel order but places each pixel's R, G, and B values next to one another. We considered both encodings to check whether this one dimensional ordering made a difference to next-byte prediction performance, but we did not find any appreciable effect.

### B.3 RAW AUDIO: NATURAL SPEECH

Our raw-audio benchmarks use the official Speech Commands v0.02 test archive, released under CC BY 4.0. It contains 4,890 one-second mono recordings, encoded in WAV files, composed by ten target commands together with unknown-word and silence examples.

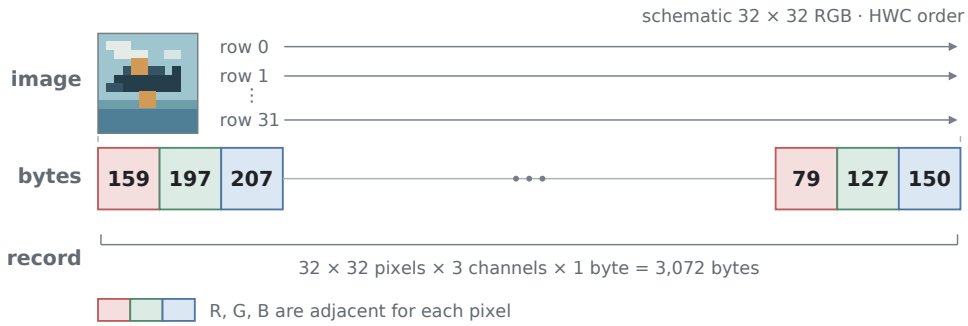



Figure 8: Illustration of the interleaved (HWC) encoding, in which the RGB values of each pixel are adjacent. The thumbnail and displayed byte values are schematic.

A WAV file mixes the waveform with a header, and its samples are signed 16-bit little-endian values. We discard the header, paths, category labels, and other metadata. Each amplitude $s$ is clipped to the PCM16 range and mapped to the unsigned byte $\lfloor (s + 32768)/256 \rfloor$, so zero amplitude becomes 128 and neighboring bytes remain neighboring points in time.

We provide three versions of the same recordings. The 16 kHz version quantizes the original samples directly. For the 8- and 4 kHz versions, fixed anti-aliasing filters downsample each recording independently by factors of two and four before the same quantization. These benchmarks test short-time acoustic prediction. With context length 256, each record contains 255 consecutive waveform bytes. The 255 bytes span 15.94 ms at 16 kHz, 31.88 ms at 8 kHz, and 63.75 ms at 4 kHz. Reducing the sample rate sacrifices high-frequency detail but exposes a longer interval to the same bounded-context model.

### B.4 MUSIC

Raw audio is physically direct but temporally expensive. PCM8 sampled at 16 kHz consumes 16,000 bytes per second; even the repository's 4 kHz PCM8 variant consumes 4,000 bytes per second. So even a 4096 context length is only able to understand local features. Symbolic score music representation is instead much more compact. At four bytes per quarter note, 255 bytes represent 63.75 quarter-note grid units. They may contain several phrases or a substantial portion of a movement rather than a fraction of one acoustic event.

To construct this benchmark, we used the Mutopia project data. The Mutopia Project is a volunteer collection of open sheet music written in LilyPond and based on editions in the public domain contributors. Its contribution pages provide downloadable notation, PDF, and MIDI artifacts together with source, maintainer, typesetting, and license .

From the full Mutopia project, we took a selection of 40 highly recognizable Western classical pieces under Public Domain license, including familiar pieces by Beethoven, Mozart, Bach, Chopin, Debussy, Schubert, Tchaikovsky, and others. A Standard MIDI File serializes a technical event stream rather than a direct sequence of musical states. It can contain a header, format and track declarations, variable-length delta encodings, tempo events, time signatures, program changes, channels, velocities, controllers, text, copyright notices, names, and end-of-track events. Two files representing substantially the same score can differ in many raw bytes because of exporter, track layout, event ordering, and metadata choices. We stripped all this data, retaining only the melodies.

To convert the complex music information in the MIDI melodies into a simple "time-sequence" byte encoding, we quantized the scores in 16th notes, each byte in the benchmark corresponding the state of that time grid cell: either a new note, a continuation of the previous one, or a silence. The selected track need not be monophonic. The benchmark creates a monophonic output by choosing at most one active source note in every grid cell.

We use byte value 0-127 to encode the absolute MIDI pitch . MIDI pitch is an absolute semitone number: 60 is middle C, 61 is C-sharp/D-flat, 62 is D, 63 is D-sharp/E-flat, 64 is E, and 67 is G. Byte

value 128 represents continuing holding the previous note in the current byte cell, 129 represents a pause, and 130 denotes the end of a piece. In this benchmark, the bytes 131-255 are unused.

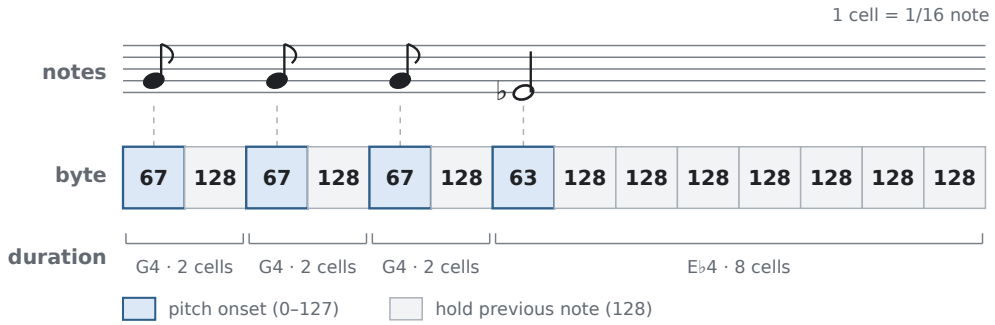



Figure 9: Qualitative illustration of the melody byte encoding using the opening motif of Beethoven's Fifth Symphony. Each cell is one sixteenth note: bytes 0–127 start a pitch, while byte 128 holds the previous note.

Together with the benchmark data, we provide MIDI files reconstructed from this simplified byte encoding, which we used to check the main themes were still recognizable.

We truncated the longer pieces to 4095 bytes.

### B.5 DNA

To construct this benchmark, we used the large DNA dataset released with the KoLMogorov Test Yoran et al. (2025). The original test asks code-generating models to produce short programs that reproduce a sequence exactly. Here we reuse only its released DNA sequences for ordinary next-symbol prediction, and refer to the resulting benchmark as `kolmogorov_dna`.

The KoLMogorov paper describes this stream as derived from GRCh38, the curated human reference assembly NCBI. A reference assembly is a composite sequence assembled and maintained as a common coordinate system; it is not a file of raw sequencing reads or the observed genome of one person. The source retains upper- and lower-case versions of the four bases A, C, G, and T as eight distinct symbols. Lower-case sequence is commonly used for soft masking, which marks repetitive or low-complexity regions while retaining the underlying base. The released `dna.bin` contains the numeric values 0-7, using the encoding

$$0 \mapsto \texttt{a}, \quad 1 \mapsto \texttt{c}, \quad 2 \mapsto \texttt{t}, \quad 3 \mapsto \texttt{g}, \quad 4 \mapsto \texttt{A}, \quad 5 \mapsto \texttt{C}, \quad 6 \mapsto \texttt{T}, \quad 7 \mapsto \texttt{G}.$$

From the large release, we retain its first 32 MiB numeric symbols. We divide this prefix without overlap into 131,586 consecutive records of 255 symbols, discarding the final tail. A predictor told that only eight values are possible could obtain 3 BPB by assigning them equal probability, whereas a uniform prediction over the learner's full 256-byte output space costs 8 BPB. The learner must therefore recognize the compact alphabet as well as exploit local base composition, masking runs, repeats, and motifs visible within 255 symbols.

### B.6 Formal mathematics: Metamath

To construct this benchmark, we used a fixed revision of `set.mm`, the main Metamath database Metamath contributors. Metamath provides a simple language for writing and checking mathematical proofs. The database contains declarations of symbols, hypotheses, axioms, and theorem statements with their proofs. Proofs refer to earlier statements by their labels and are stored in a compressed form, using compact character strings to encode the proof steps.

We remove the comments delimited by `$(` and `$)`, including explanatory prose, and discard empty lines. Within each remaining line, we replace runs of whitespace by a single space, remove leading and trailing whitespace, and end the line with one newline byte. We retain the formal content in its source order, including the statement labels, formulas, and compressed proofs. The resulting text is

represented directly by its ASCII byte values, with spaces and newlines included in the sequence.

With context length 256, we divide this stream without overlap into 143,166 consecutive records of 255 bytes, discarding the final 176 bytes. The windows can cross line, statement, and proof boundaries. The predictor therefore has to exploit regularities such as recurring syntax, formula fragments, labels, and patterns in the compressed proofs through the same byte interface used for every other benchmark. This tests next-byte prediction of formal mathematical text; the model is not asked to construct or verify a proof.

### B.7 C SOURCE CODE

To construct this benchmark, we used file B from the Algorithmic Information Theory Data Compression Challenge (AITDCC) AITDCC. The challenge compares lossless compressors across different kinds of data. Here we reuse its released C source for next-byte prediction. We extract the complete file from a fixed revision of the official AITDCC archive; it contains 1,168,767 bytes of source text.

The source is ASCII text, and we retain its original byte values. Comments, copyright notices, preprocessor directives, identifiers, and literals remain, together with indentation, tabs, spaces, blank lines, and newlines. We do not parse or compile the source, normalize its formatting, or use a C-specific tokenizer. The predictor receives the original text as one continuous byte stream.

With context length 256, we divide this stream without overlap into 4,583 consecutive records of 255 bytes, discarding the final 102 bytes. The windows follow byte positions and can cross line, statement, function, and source-file boundaries. The predictor therefore has to exploit regularities such as C syntax, recurring identifiers, comments, and formatting through the same byte interface used for every other benchmark. Repeated names and code patterns provide structure beyond individual characters, while comments also retain the spelling and local syntax of natural language.

## C EMERGENT MATHEMATICAL STRUCTURE

During self-play, the generator discovers programs whose output tapes exhibit recognizable mathematical structure. We search the generated programs from our model-scaling experiments for five families of sequences: arithmetic, quadratic, and cubic sequences; Fibonacci-like sequences; and geometric sequences. Generated programs from the self-play runs were retained only once every 256 training rounds, so discovery times can be measured only at this 256-round resolution. In contrast, the uniform-sampling baseline described below checks programs at every round. Thus the performance gap between self-play and the random baseline is likely even wider, as the reported self-play discovery rounds are therefore conservative upper bounds on the true first occurrence of each structure. Table 2 summarizes the observed families and compares their discovery times with uniform program sampling.

**Detection criteria.** We allow up to 30 unrelated leading bytes before the structured portion of a tape begins. The remainder of the tape must satisfy the corresponding recurrence modulo 256. Arithmetic, quadratic, and cubic sequences are defined by constant first, second, and third finite differences, respectively, with a sequence assigned to the lowest-order family it satisfies. Fibonacci-like sequences satisfy

$$v_n = v_{n-1} + v_{n-2} \pmod{256},$$

for an arbitrary seed pair, while geometric sequences satisfy

$$v_n = r v_{n-1} \pmod{256}$$

for some integer ratio $r$.

To eliminate degenerate matches, we additionally require a minimal period of at least 30, evaluated both over the full matched region and over its trailing window. This excludes, for example, sequences with a structured transient followed by a constant tail.

**Comparison with uniform program sampling.** To estimate how readily the same structures would be discovered without self-play, we sample programs by uniformly sampling the primitive

| Family (mod 256) | Example program | Its output | Earliest round | $\mathbb{E}$[first round] (univ. prior) |
|---|---|---|---|---|
| Arithmetic | `S+[.++]` | $1, 3, 5, 7, 9, \ldots$ | 0 | $\approx 105$ |
| Fibonacci | `S,[[.C>.C>]` | $1, 1, 2, 3, 5, \ldots$ | 512 | $> 53{,}000$ |
| Geometric | `S+[.L>]` | $1, 3, 9, 27, 81, \ldots$ | 256 | $> 53{,}000$ |
| Quadratic | `S,.[<C>>VX<RX++]` | $9, 25, 59, 111, \ldots$ | 512 | $> 53{,}000$ |
| Cubic | `S+[[-.L>L>-]-]` | $0, 254, 236, 74, \ldots$ | 512 | $> 53{,}000$ |

Table 2: Mathematical sequence families discovered in our scaling experiments. Self-play programs were retained once every 256 rounds, so the reported discovery rounds are the earliest saved rounds at which each family was observed.

augmented alphabet until an "F" symbol is drawn. We draw $1.64 \times 10^8$ programs in total. A sample is counted as a hit whenever its output satisfies the same family-level detection criterion above; it need not match the example program shown in Table 2.

For comparison with the scaling experiments, we group samples at the same rate of 1,024 newly generated programs per round. Unlike the self-play analysis, however, the uniform baseline is checked at every round rather than only once every 256 rounds. The comparison therefore underestimates the gap.

The arithmetic family occurs 1,526 times in the uniform baseline, giving an estimated probability of $9.3 \times 10^{-6}$ and an expected first-discovery round of approximately 105 (95% CI: 100–111).

We observe no Fibonacci, geometric, quadratic, or cubic matches in the $1.64 \times 10^8$ baseline samples. By the rule of three, this gives the one-sided 95% bound

$$p \leq 1.8 \times 10^{-8},$$

corresponding to an expected first-discovery round greater than 53,000 at 1,024 programs per round. Despite the coarser observation schedule for self-play, all four families are observed there by round 512, compared with no occurrences in more than 53,000 rounds' worth of uniformly sampled programs. Because all four families have zero observed baseline hits, however, the sampling experiment establishes only a common lower bound on their rarity and does not determine their relative frequencies.

## D ICL TASKS

Each ICL task has the form

$$\big[0, x_1^1, \ldots, x_k^1, f(x_{1\ldots k}^1)\big], \ldots, \big[0, x_1^m, \ldots, x_k^m, f(x_{1\ldots k}^m)\big], \big[0, x_1^{m+1}, x_2^{m+1}, \ldots, x_k^{m+1}, \bullet\big]. \quad (14)$$

Each example is prepended by a sentinel '0' byte and is followed by the $k$ inputs bytes and then the function applied to those bytes (brackets are for visual separation and do not appear in the actual byte sequence). After $m$ examples are shown the next example sentinel and input are passed to the model which fills in its prediction. The score for the model is the probability that it predicts the correct token, $f(x_{1\ldots k}^{m+1})$.

We take $f$ to range over the tasks

1. REVERSE STRING: Given a word $x_1 x_2 \cdots x_k$ reverse it. $f(x_1, x_2 \ldots x_k) = (x_k, x_{k-1}, dots x_1)$
2. Stack: given a series of stack operations return the result after a final pop. Stack operations are either push, or pop, sampled with equal probabilities when the stack height is at least one. Push and pop are represented by the bytes 250 and 251 respectively. The byte after is either the argument for push, or the result for pop. Example $250, 1, 250, 2, 251, 2, 250, 3, 251$ would be followed by 3.
3. ASSOCIATIVE RECALL is a dictionary association task. The dictionary has size $V$ and maps bytes from $1 - 255$ to bytes from $1 - 255$ (repetition allowed). The dictionary is first printed in the form of key-value pairs in the format of eq. (14) so that all pairs are visible to

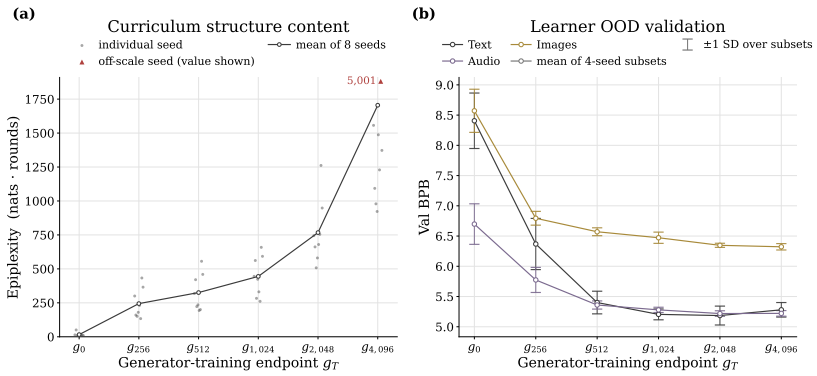



Figure 10: **Later generator checkpoints provide additional, non-redundant training value.** For each generator-training endpoint $T > 0$ we build a fixed corpus of programs by sampling from 16 generator snapshots spaced $T/16$ apart and ending at $T$; $g_0$ uses the untrained generator. A 1M-parameter learner is then trained from scratch on each corpus, four seeds per endpoint. **(a)** Epiplexity, the excess training loss the learner accumulates before converging, grows steadily with $T$: corpora written by later checkpoints contain more structure. **(b)** Out-of-distribution validation BPB of 4 ensembled seeds on text, audio, and images falls with $T$. Together, the two panels show that later checkpoints enrich the curriculum rather than repeating earlier material, and that this additional data improves transfer to unseen datasets.

the model. Evaluation proceeds in a similar manner, with random key-value pairs sampled and presented to the model. Pairs are drawn at random after the initial print and do repeat.

4. SUM is the task of summing two bytes *mod* 256. The format is $[0, x_1, x_2, (x_1 + x_2)\ \textit{mod}\ 256]$. $x_1, x_2 \neq 0$.
5. MAX / MIN are the task of finding the min and max over the $k$ input bytes. That is $f(x_1, \ldots x_k) = \min(x_1, \ldots x_k)$ and similarly for max.

**The generator produces increasingly useful training data.** We next ask whether the generator improves over the course of self-play. For each endpoint $T$, we construct a fixed corpus $\mathcal{D}_T$ by sampling 4.19M programs uniformly across 16 generator checkpoints up to $T$ (with $g_0$ as untrained), and train a fresh 1M-parameter learner for one epoch on each corpus under a fixed token budget.

We evaluate each corpus by its *epiplexity*—the amount of structure extractable by a compute-bounded learner (Finzi et al., 2026)—and by zero-shot performance on held-out text, audio, and images (Figure 10). Both improve steadily with $T$: later generators produce data containing more learnable structure and yielding better transfer, indicating that self-play continually improves the quality of the training distribution. Consistent with this, the generator also discovers recognizable mathematical sequences far earlier than expected under the fixed universal prior (table 2).

## E BRAINF*CK DETAILS

The generator produces programs in a minimal Turing complete language. We use a Brainf*ck[3]-like universal machine, similar to the variant introduced in Grau-Moya et al. (2024), whose eight single-character instructions move the head between cells (`<`, `>`), increment or decrement the cell under it (`+`, `-`), loop (`[`, `]`), and read or write bytes (`,` and `.`).

Programs are strings over the alphabet $\mathcal{A} = \{$`<`, `>`, `+`, `-`, `[`, `]`, `.`, `,`, `F`$\}$ of this machine $U$, where `F` is an explicit end-of-program token. A program $x \in \mathcal{A}^{\leq L}$ is executed on $U$ under a bounded step and memory budget, with tape cells taken modulo a fixed modulus $m = 256$; the input instruction (`,`) reads i.i.d. uniform random bytes from a random tape $\omega$, so execution defines an output distribution

[3]For completeness, this stands for Brainfuck.

per program. The output

$$y \;=\; U(x,\omega) \;\in\; \{0,\ldots,m-1\}^{T}$$

is the sequence of the first $T$ bytes the program emits, zero-padded if it halts early.

As an example, the following program implements a three-iteration `for` loop, using its first cell as the loop counter and emitting its second cell once per iteration:

$$\underbrace{\texttt{+++}}_{\text{cell}_0 \leftarrow 3} \quad \underbrace{\texttt{[}}_{\text{while cell}_0 \neq 0} \quad \underbrace{\texttt{>+.}}_{\text{cell}_1 \leftarrow \text{cell}_1 + 1;\ \text{emit cell}_1} \quad \underbrace{\texttt{<-}}_{\text{cell}_0 \leftarrow \text{cell}_0 - 1} \quad \underbrace{\texttt{]}}_{\text{end loop}} \quad \underbrace{\texttt{F}}_{\text{halt}}$$

It emits $y = (1, 2, 3, 0, \ldots, 0)$: one byte per iteration, then zero-padding once the program halts.

Importantly, *every* string over $\mathcal{A}$ is executable. There is no syntax error: the only way a program can be malformed is through unmatched brackets, which we treat as no-ops. In practice we cannot implement an unbounded tape or unbounded run times, so our machine has finite memory and finite time budgets; wherever a program exceeds one of these bounds, the semantics are defined to wrap or halt rather than fault. The tape is circular, so that a head move off either end of the finite memory wraps around, and incrementing or decrementing a cell wraps modulo $m$. As a consequence, there are no out of bounds errors. Execution always terminates — at the step budget, the end of the program, or the $T$-th emitted byte, whichever comes first.

The machine as described is Turing complete using the eight Brainf*ck instructions alone. In practice, however, common patterns such as clearing a cell, moving a value to a neighbor, or scanning to the next zero cell are frequently used in human written Brainf*ck programs, and appear to increase the efficiency of self-play in preliminary experiments. We therefore extend the alphabet with the ten single-character tokens in table 3. All methods we compare draw from this same augmented alphabet, including the Solomonoff-prior baseline in section 3.1 and the uniform-sampling comparison, so the added primitives cannot account for any difference between self-play and its controls. Several of the discovered program families in table 2 use these tokens.

| Token | Expansion | Effect on tape |
|---|---|---|
| Z | `[-]` | Clear current cell |
| R | `[->+<]` | Clear and add $x$ into right neighbor |
| L | `[->+++<]` | Clear and add $3x$ into right neighbor |
| N | `[-<->]` | Clear and subtract $x$ from left neighbor |
| C | `[->+>+<<]` | Clear and add $x$ into the two right cells |
| G | `[>]` | Scan right to next zero cell |
| H | `[<]` | Scan left to next zero cell |
| W | `[[-]>+<]` | If current $\neq 0$: increment right and clear current |
| V | `[.>]` | Print stored string until a zero cell |
| X | `[-]++++++++++++++++` | Set cell to 16 |

Table 3: The set of characters used to augment the Brainf*ck language, their expansion in terms of pure Brainf*ck, and their meaning. $x$ corresponds to the value at the current memory cell.

## F Reward Ablations Table

To support our choice of reward we show several ablations along with variants of the real reward. The ”none” reward is the canonical setup, which as we have seen is already better than the ”uniform” ablation where we set the generator to sample program tokens uniformly from the alphabet. Additionally we consider a signed version of the reward which does worse than the absolute value, which shows that the absolute value is useful. When we shuffle the reward between programs of the same batch we break the correlation, and see a much worse result demonstrating that the mere marginal distribution of rewards is not sufficient to drive progress.

The last step reward is evaluating against the one-step weight difference rather than taking a window back to $e/2$ rounds, which is clearly worse, demonstrating that there is some value to averaging over larger blocks. Additionally the loss delta, which is putatively similar to the last step reward (to first order) is worse still, showing that the first-order reward is usefully more informative than the finite

| | ablation | | | | | | |
|---|---|---|---|---|---|---|---|
| dataset | None | uniform | signed | shuffle | last_step [b] | loss_delta [a,b] | negate |
| text (dclm) | 5.34 | 7.75 | **5.06** | 5.96 | 6.39 | 7.40 | 10.62 |
| Metamath | **3.38** | 7.39 | 3.47 | 4.39 | 4.34 | 6.46 | 10.52 |
| C source | 4.16 | 7.92 | **4.10** | 4.79 | 4.95 | 6.37 | 10.60 |
| DNA (8-symbol) | **2.29** | 3.02 | 2.45 | 2.50 | 3.22 | 3.57 | 8.37 |
| arithmetic | **0.22** | 7.94 | 0.51 | 0.73 | 1.26 | 1.92 | 10.64 |
| audio 8-bit PCM | **2.27** | 3.93 | 2.45 | 2.91 | 3.39 | 3.67 | 7.67 |
| audio 16-bit PCM | **5.02** | 6.08 | 5.24 | 5.79 | 5.97 | 6.32 | 9.02 |
| melody (Mutopia) | **2.20** | 7.09 | 2.21 | 3.26 | 3.35 | 4.14 | 11.00 |
| CIFAR-10 (planar) | **5.96** | 7.83 | 6.14 | 7.14 | 7.22 | 7.59 | 10.59 |
| random bytes | **8.02** | **8.02** | 8.05 | **8.02** | 8.29 | 8.61 | 10.57 |

Table 4: **Reward ablations of the self-play generator** at the 1M parameter model. Each entry shows the validation loss in bits per byte of the 4-seed ensemble on 256 held-out sequences per dataset, scored from the final-round checkpoints. Every ablation changes exactly one property of the canonical reward $r_i = |\langle P \odot \nabla L(y_i; \theta_{\text{now}}), \theta_{\text{past}} - \theta_{\text{now}} \rangle|$ with $\theta_{\text{past}} = \theta_{\lfloor e/2 \rfloor}$ (*None*). *Signed* drops the absolute value, *shuffle* permutes the rewards across the program pool, *last_step* uses the one-step window $\theta_{\text{past}} = \theta_{e-1}$, *loss_delta* uses the realized progress $L_i(\theta_{\text{pre}}) - L_i(\theta_{\text{post}})$, *negate* flips the reward's sign, and *uniform* removes the generator entirely (i.i.d. uniform programs). On the majority of datasets, the canonical reward is best, or nearly best, and it often has lower variance as well.
[a] One loss_delta seed stopped at round 2587 and is excluded; its ensemble is over 3 seeds.
[b] last_step and loss_delta are bimodal across seeds (e.g. dclm single-seed 6.4 / 6.5 vs 10.6 / 11.1 for last_step; 7.0 vs 9.8 / 12.3 for loss_delta); their ensembles average over the diverged seeds.

difference.

Finally the negative of the reward shows performance worse than a randomly initialized model, indicating that the reward does prefer systematically better programs to worse ones across the board.

## G POOL CONSTRUCTION

At round $e$, we construct a fixed pool of programs

$$\mathcal{B}_e = \mathcal{B}_e^{\text{fresh}} \dot{\cup} \mathcal{B}_e^{\text{mut}} \dot{\cup} \mathcal{B}_e^{\text{replay}}.$$

The fresh pool $\mathcal{B}_e^{\text{fresh}}$ consists of the latest programs sampled from the generator $g_\phi$. A fraction of on-policy samples is replaced by $\mathcal{B}_e^{\text{mut}}$: single-token substitutions, insertions, or deletions of positively rewarded programs drawn from the quality-diversity bank. The replay pool $\mathcal{B}_e^{\text{replay}}$ is formed by drawing programs uniformly without replacement from the bank of non-replay programs produced in earlier rounds. These programs are re-executed with fresh random tapes. Mutations improve local exploration by making small edits to promising programs and replay mitigates catastrophic forgetting of behaviors discovered in earlier rounds. For mutation, we use a MAP-Elites-style algorithm (Mouret & Clune, 2015), so that mutation does not concentrate exclusively on the highest-reward programs. Concretely, each program is assigned to a niche according to two descriptors: (i) its maximum dynamic loop depth, binned from $0$ through $8$ with larger depths clamped to the final bin, and (ii) its program-body length, bucketed at $8$, $16$, and $32$ tokens. This produces at most $36$ niches in total. Only programs receiving positive generator reward are admitted to the archive, and within each niche we retain the top $8$ programs according to their stored reward. Because the usefulness of a program depends on the learner's current state, stored rewards are decayed by a factor of $0.97$ each round, allowing newly useful programs to replace stale elites. Mutation parents are selected uniformly across occupied niches, rather than uniformly across all archived programs, preserving representation for rarer structural behaviors such as deeply nested programs.

## H RANDOM-PCFG PRETRAINING

**Grammar sampling.** Each grammar $G = (V, \Sigma, R, S)$ is drawn as follows. The terminal set $\Sigma$ is a uniform sample, without replacement, of $n_\Sigma \sim \mathcal{U}\{2, \ldots, 16\}$ distinct byte values from $\{1, \ldots, 255\}$ (byte 0 is reserved as padding and never emitted). The grammar has $|V| \sim \mathcal{U}\{1, \ldots, 8\}$ non-terminals with start symbol $S = V_0$. Each non-terminal receives $\mathcal{U}\{1, \ldots, 4\}$ productions; production probabilities are i.i.d. $\mathcal{U}(0, 1)$ weights (plus $10^{-6}$), normalized to sum to one. Each right-hand side contains $\mathcal{U}\{1, \ldots, 4\}$ symbols, each independently a terminal with probability $p_T = 0.5$ (uniform over $\Sigma$) and otherwise a uniform non-terminal. *Productivity repair:* if a non-terminal ends up with no terminal-only production, the right-hand side of one uniformly chosen production is replaced by a freshly sampled terminal-only string (its probability unchanged), so every non-terminal can terminate in one step.

**Derivation and row packing.** A word is derived by leftmost expansion with an explicit stack, sampling productions by their probabilities, and stops when the stack empties, the output reaches 64 bytes, or $10^4$ expansions elapse (a backstop for grammars with unbounded expected yield); if expansion produces no terminal, the start symbol's stored one-step terminal yield is emitted, so every word has $\geq 1$ byte. For each 4,095-byte training row, *one* fresh grammar is sampled and words are derived from it and concatenated until the row fills (the final word is truncated). Rows therefore contain repeated material from a single small random grammar and are zero-free by construction.