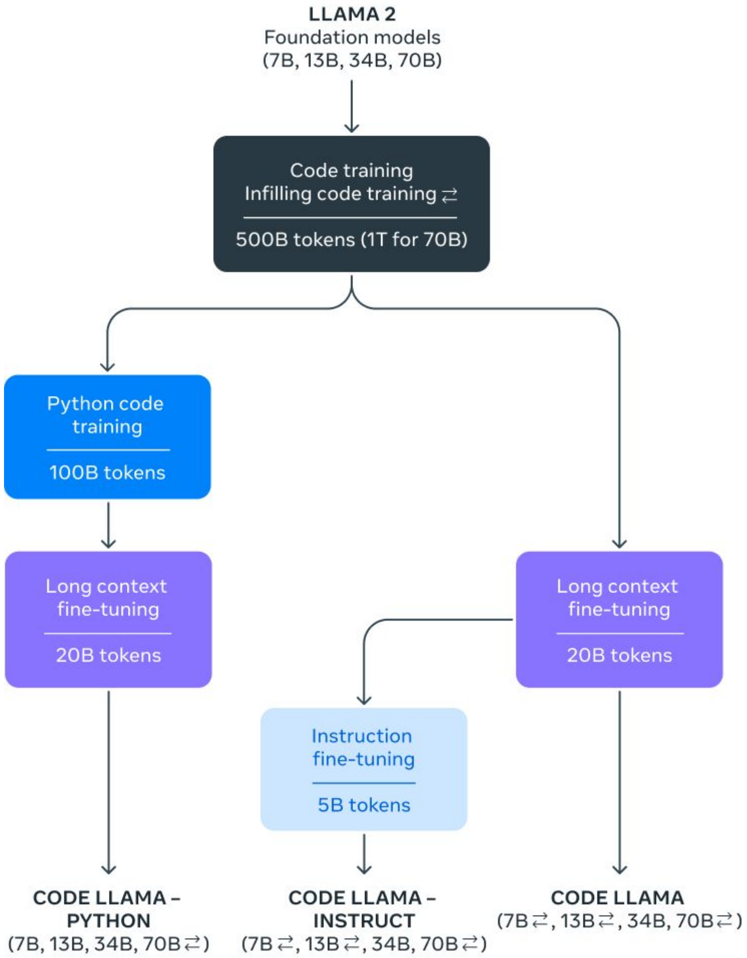
LLMs for code (3)

Marc LELARGE
INRIA-ENS
Paris
Co-teacher: Nathanaël FIJALKOW

with slides from Sean Welleck
https://wellecks.com/

Pretraining and scaling laws
Example: CodeLlama

- Pretraining
  - 2 trillion (T) tokens of mixed data (web, code ...) to train Llama 2
- Adaptation
  - Continued pretraining (500 billions B tokens of mostly code data)
  - Finetuning: long context, Python code, instructions



Learning: maximum likelihood = next token prediction

Make observed data likely under the model: maximum likelihood

$$\arg\max_{\theta} \frac{1}{|\mathcal{D}|} \sum_{(y_1,\ldots,y_\ell)\in\mathcal{D}} \log p_\theta(y_1,\ldots,y_\ell) = \arg\max_{\theta} \frac{1}{|\mathcal{D}|} \sum_{(y_1,\ldots,y_\ell)\in\mathcal{D}} \sum_{t=1}^{\ell} \log p_\theta(y_t|y_{<t})$$

Compute for performing forward and backward passes using a transformer on token sequences:

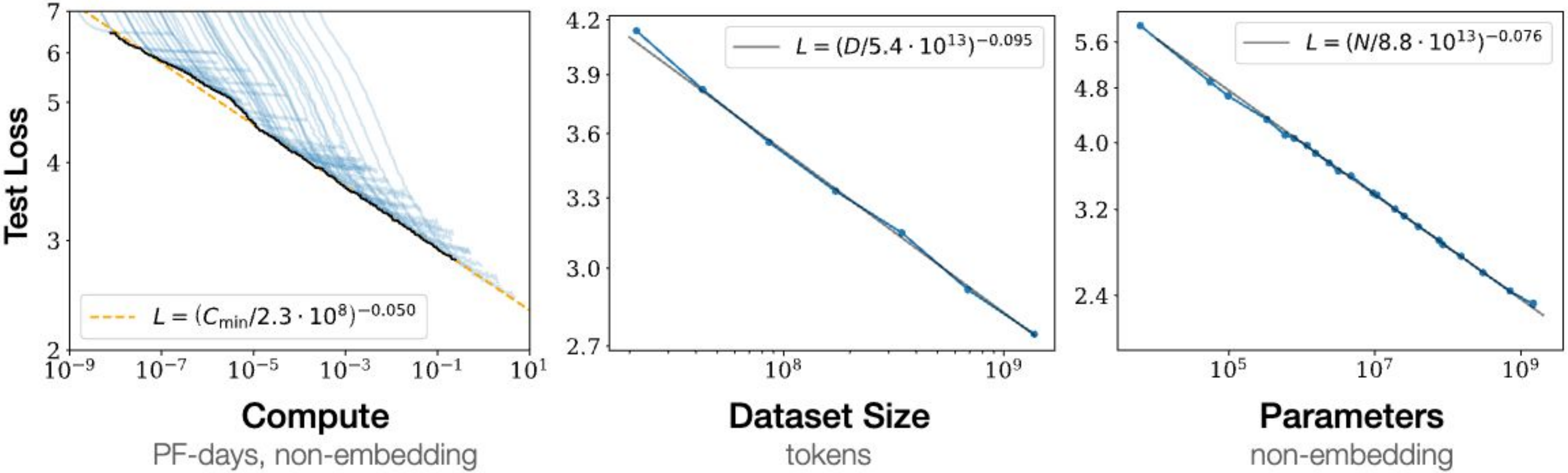
C≈6ND

with N: number of model parameters
D: number of tokens
C: compute in floating point operations (FLOPS)

Example: Llama 2

C≈ 6 x 7 Billion x 2 Trillion = 8.4 10^22 FLOPS

Loss gets better with more compute



We can increase compute by increasing N the number of parameters, training on more tokens D, or a combination thereof.

Test loss scales as a power-law with the amount of compute: $L(X) \propto X^{-a_X}$
where X is compute C, dataset size D or number of parameters N.

Lower loss translates to better task performances

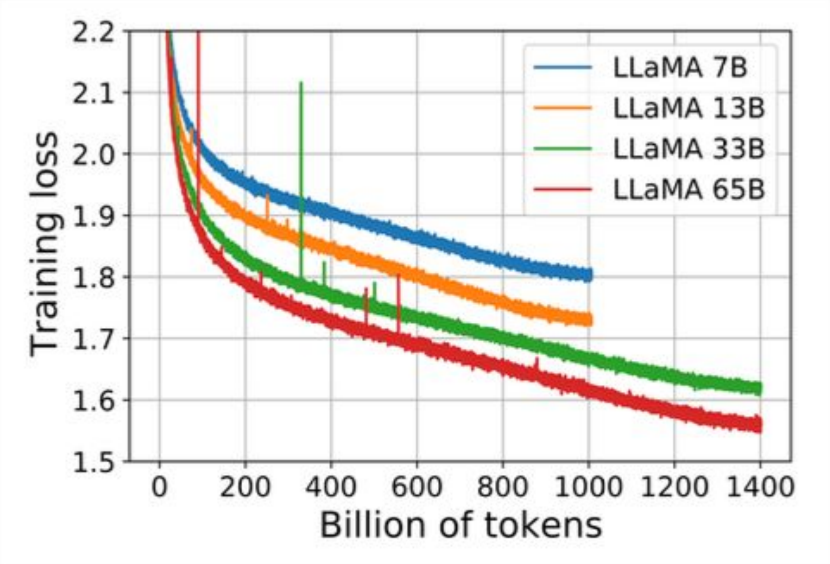


Figure 1: Llama training loss

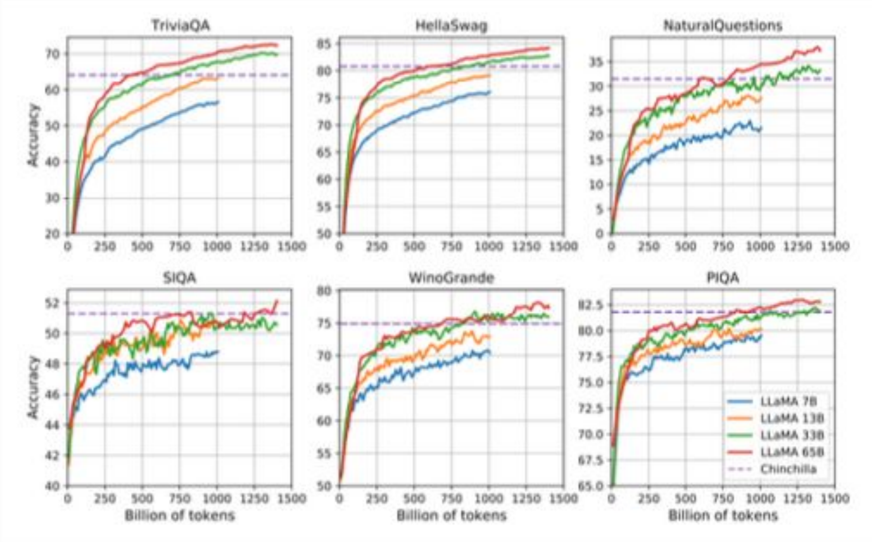
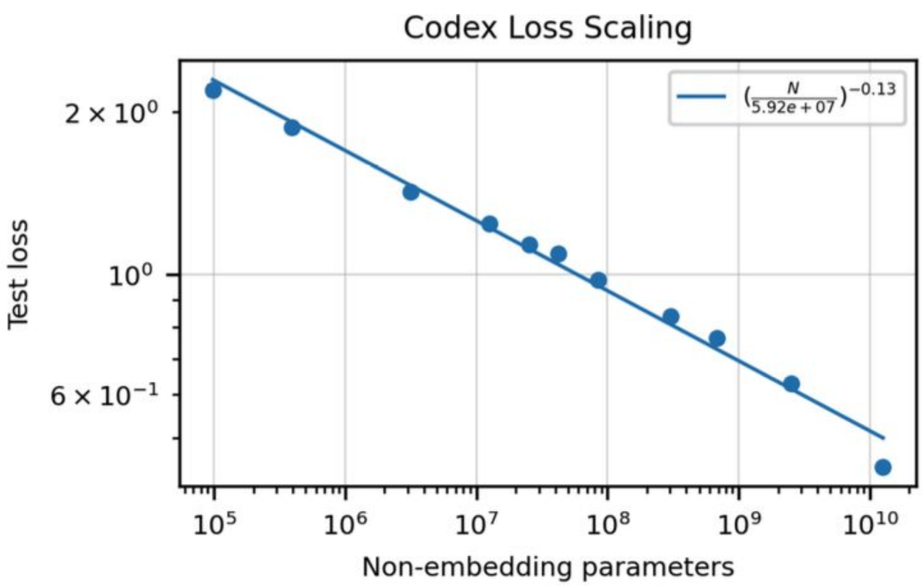
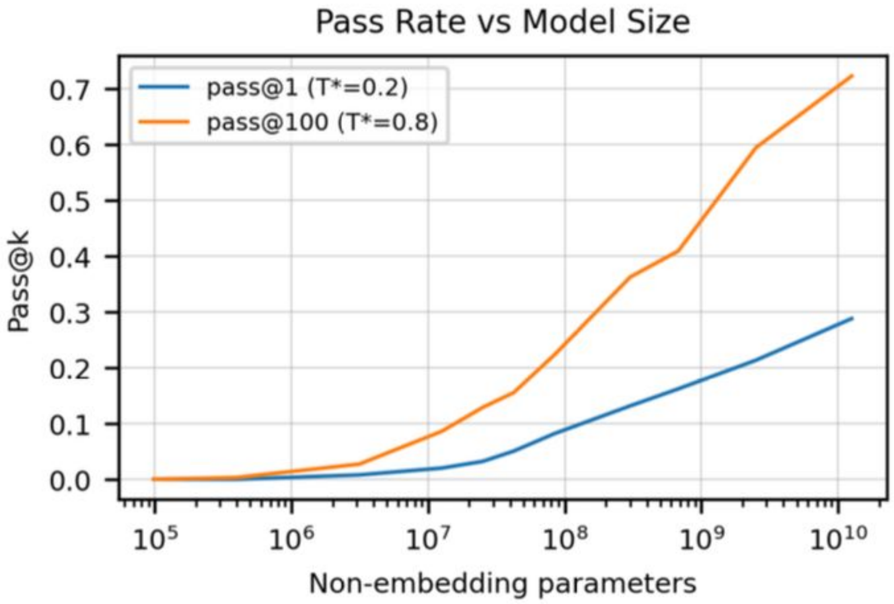


Figure 2: Llama task performance

It holds for code too!



Codex test loss scaling in number of parameters N



Codex pass rate on HumanEval as a function of parameters N

Scaling laws: allocation

- Pretraining = fitting a target distribution
- Fit gets better as we increase compute following a scaling law
- Should I spend my compute on larger model or on more data?
- **Allocation**: for compute budget C, choose number of parameters N and tokens D that minimizes loss:

$$\arg \min_{6ND<C} L(N, D)$$

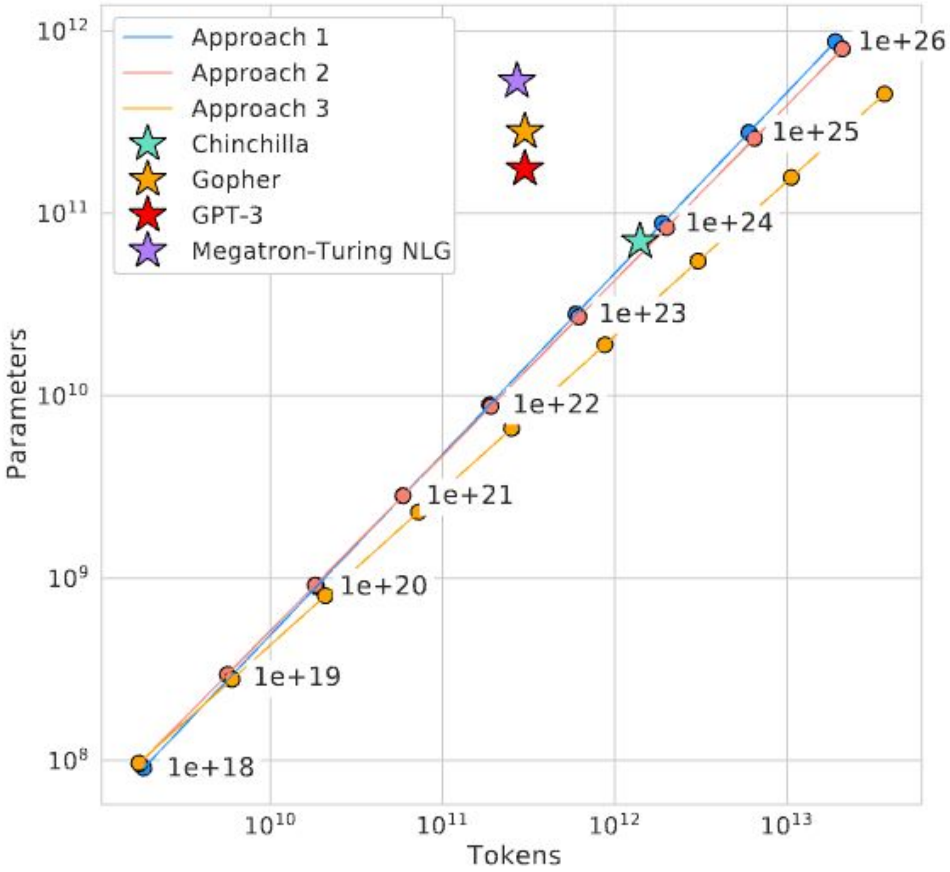
DeepMind

Training Compute-Optimal Large Language Models

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*

*Equal contributions

Allocation: Chinchilla



Optimal number of tokens and parameters for a training FLOP budget.

For a fixed FLOP budget, we show the optimal number of tokens and parameters

Allocation: Chinchilla

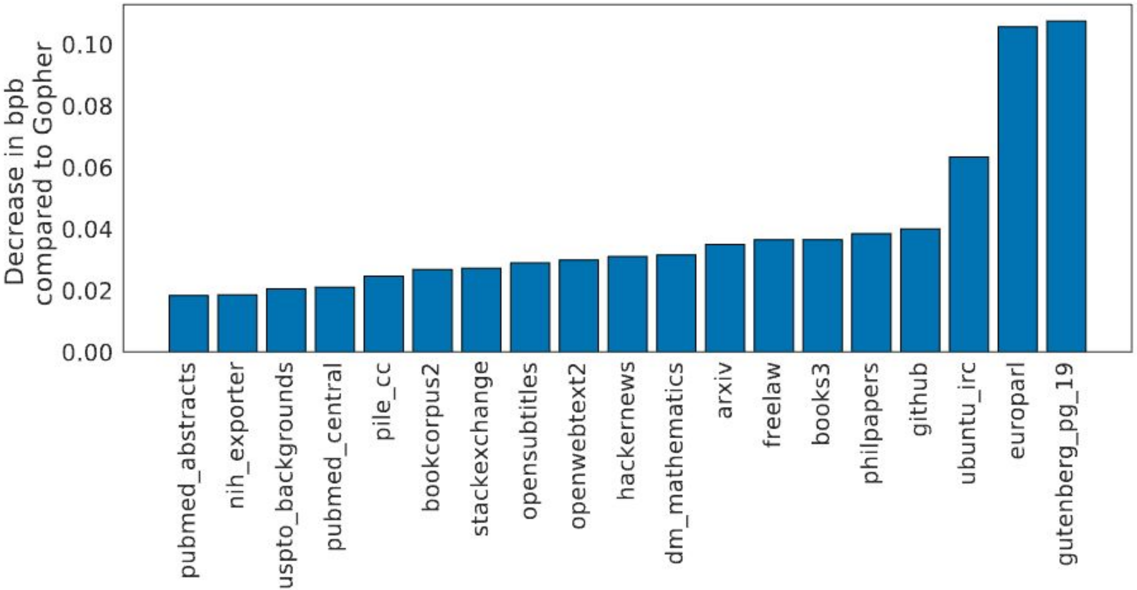


Figure 5 | **Pile Evaluation.** For the different evaluation sets in The Pile (Gao et al., 2020), we show the bits-per-byte (bpb) improvement (decrease) of *Chinchilla* compared to *Gopher*. On all subsets, *Chinchilla* outperforms *Gopher*.

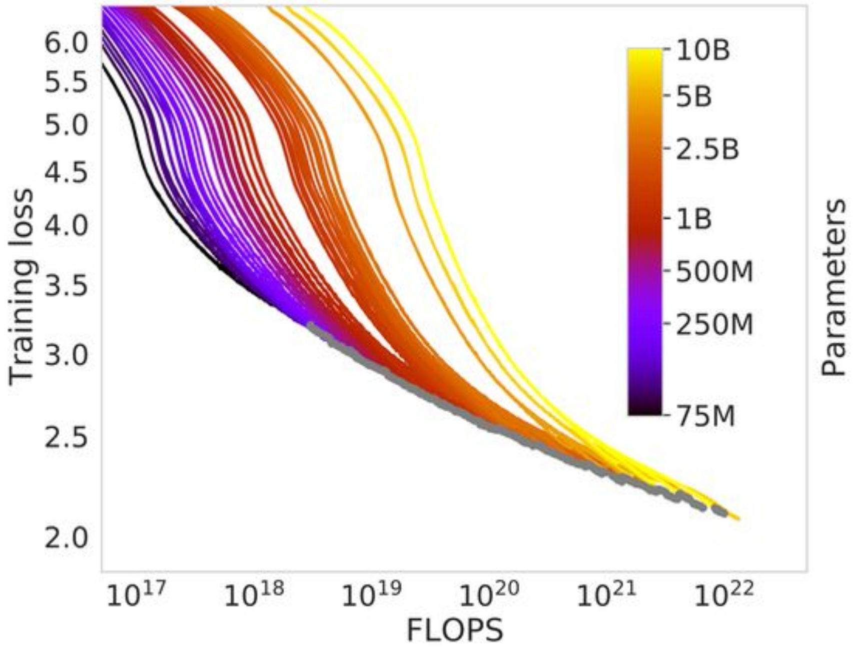
Allocation: Chinchilla

To choose Chinchilla's allocation, the authors fit scaling laws on runs with smaller amounts of compute. They used three approaches.

Approach	Coeff. a where $N_{opt} \propto C^a$	Coeff. b where $D_{opt} \propto C^b$
1. Minimum over training curves	0.50 (0.488, 0.502)	0.50 (0.501, 0.512)
2. IsoFLOP profiles	0.49 (0.462, 0.534)	0.51 (0.483, 0.529)
3. Parametric modelling of the loss	0.46 (0.454, 0.455)	0.54 (0.542, 0.543)
Kaplan et al. (2020)	0.73	0.27

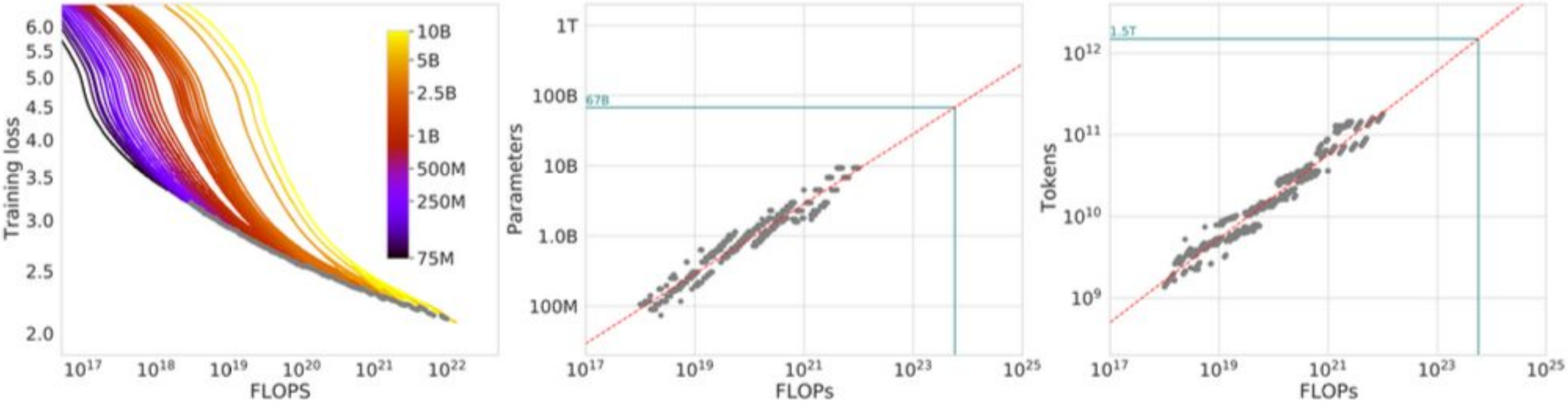
a ≈ b : parameters and tokens should be scaled at the same rate.

Approach 1: fix N and vary D



- For each size N, train many models with different number of tokens D
- For each compute C, pick the model with the lowest loss L
- We now have (C, N, D, L) examples (grey points)

Approach 1: fix N and vary D



Fit power laws using the (C, N, D, L) examples.

- Middle: optimal model size $N_{\text{opt}} \propto C^a$
- Right: optimal number of tokens $D_{\text{opt}} \propto C^b$

Allocation: Chinchilla

To choose Chinchilla's allocation, the authors fit scaling laws on runs with smaller amounts of compute. They used three approaches.

Approach	Coeff. a where $N_{opt} \propto C^a$	Coeff. b where $D_{opt} \propto C^b$
1. Minimum over training curves	$0.50\ (0.488, 0.502)$	$0.50\ (0.501, 0.512)$
2. IsoFLOP profiles	$0.49\ (0.462, 0.534)$	$0.51\ (0.483, 0.529)$
3. Parametric modelling of the loss	$0.46\ (0.454, 0.455)$	$0.54\ (0.542, 0.543)$
Kaplan et al. (2020)	0.73	0.27

a ≈ b : parameters and tokens should be scaled at the same rate.

Approach 2: IsoFLOPs curves

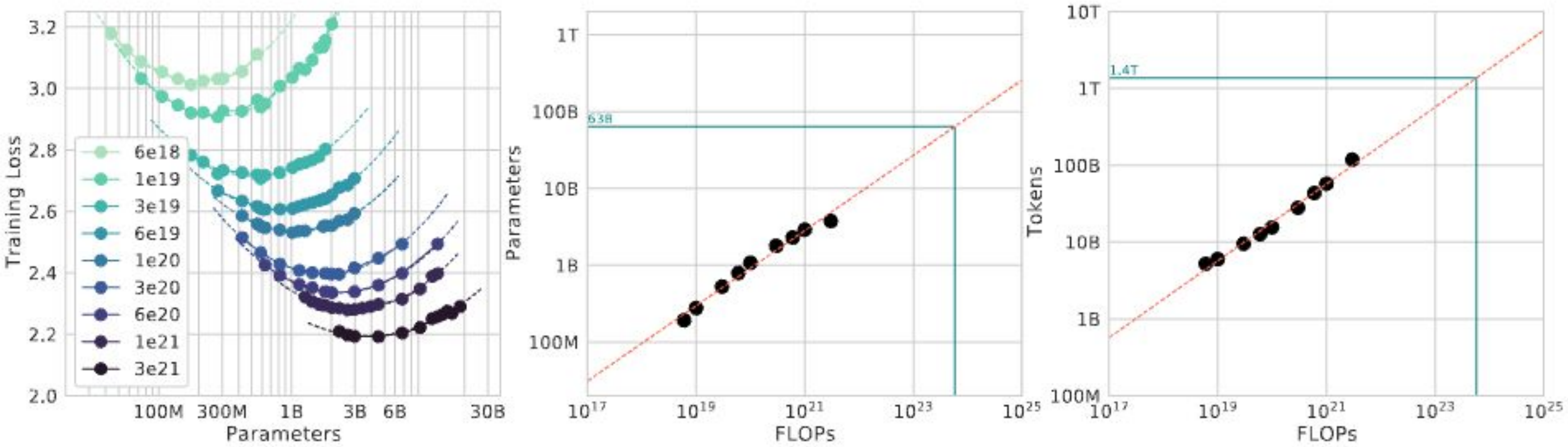


Figure 3 | **IsoFLOP curves.** For various model sizes, we choose the number of training tokens such that the final FLOPs is a constant. The cosine cycle length is set to match the target FLOP count. We find a clear valley in loss, meaning that for a given FLOP budget there is an optimal model to train (**left**). Using the location of these valleys, we project optimal model size and number of tokens for larger models (**center** and **right**). In green, we show the estimated number of parameters and tokens for an *optimal* model trained with the compute budget of *Gopher*.

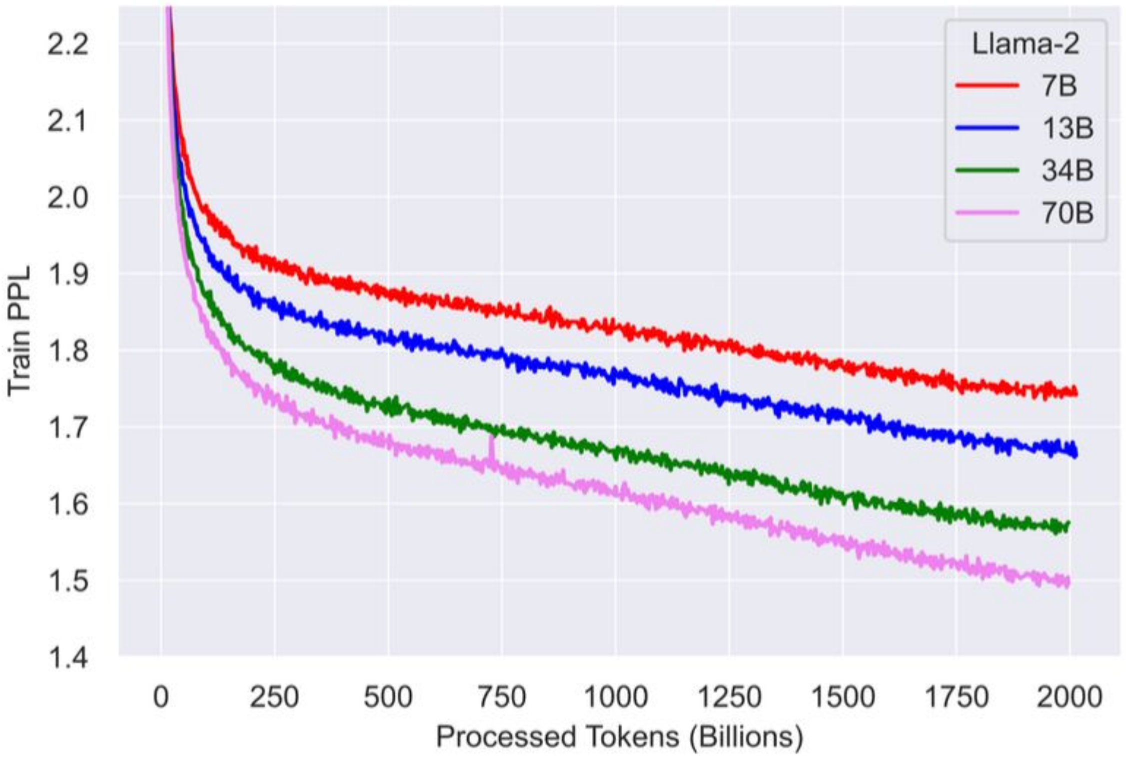
Post-Chinchilla

Scale parameters and token at a similar rate

Approach	Coeff. a where $N_{opt} \propto C^a$	Coeff. b where $D_{opt} \propto C^b$
1. Minimum over training curves	0.50 (0.488, 0.502)	0.50 (0.501, 0.512)
2. IsoFLOP profiles	0.49 (0.462, 0.534)	0.51 (0.483, 0.529)
3. Parametric modelling of the loss	0.46 (0.454, 0.455)	0.54 (0.542, 0.543)
Kaplan et al. (2020)	0.73	0.27

- The Chinchilla scaling law arguably led to a focus on scaling data
- Trend: train on even more tokens than suggested by the compute-optimal scaling law.
  Training a smaller model on more tokens may be compute optimal when inference-time compute is factored in; smaller models require less inference compute.

Post-Chinchilla



Llama 2 – more tokens than Chinchilla, equal size (70B)

Scaling laws everywhere!

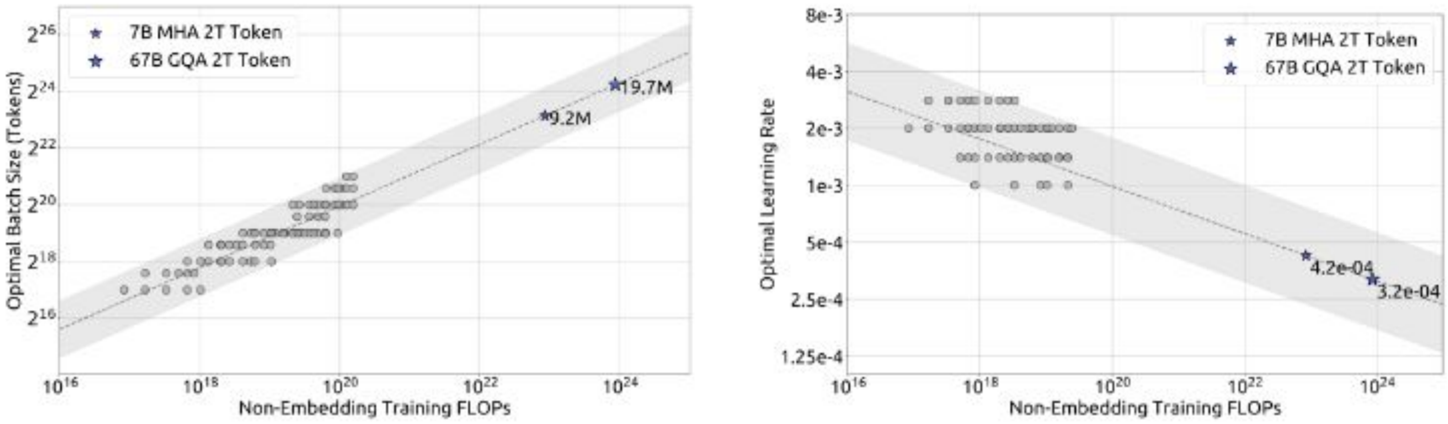
deepseek

DeepSeek LLM
Scaling Open-Source Language Models with Longtermism

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y.K. Li, Wenfeng Liang, Fangyun Lin, A.X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R.X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, Yuheng Zou [*]

[*]**DeepSeek-AI**

Scaling laws everywhere!



(a) Batch size scaling curve

(b) Learning rate scaling curve

Figure 3 | Scaling curves of batch size and learning rate. The grey circles represent models whose generalization error exceeded the minimum by no more than 0.25%. The dotted line represents the power law fitting the smaller model. The blue stars represent DeepSeek LLM 7B and 67B.

Scaling laws as a tool

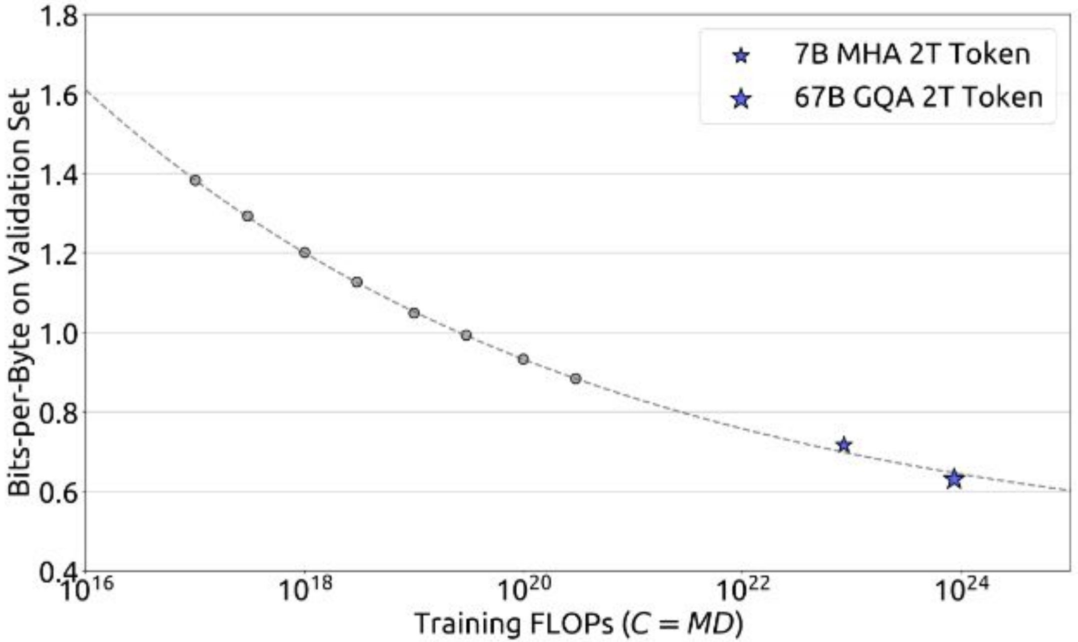
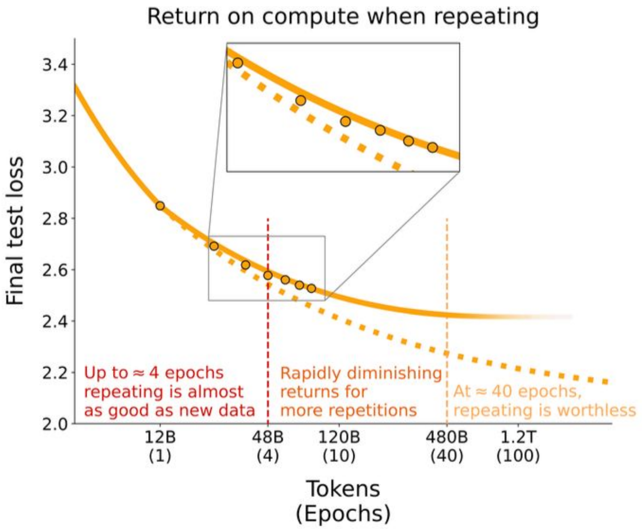
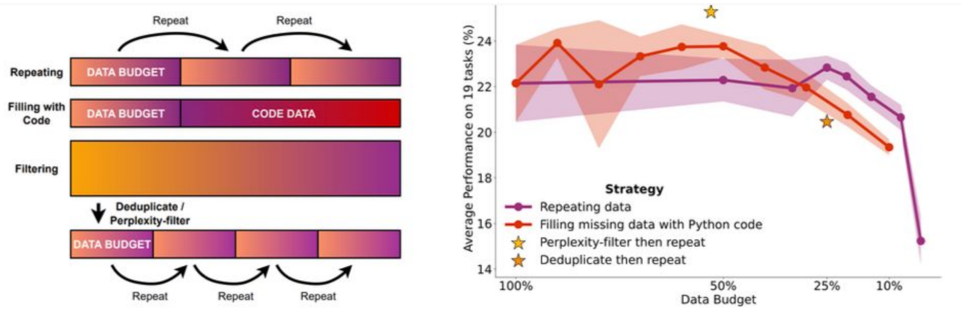


Figure 5 | Performance scaling curve. The metric is the bits-per-byte on the validation set. The dotted line represents the power law fitting the smaller model (grey circles). The blue stars represent DeepSeek LLM 7B and 67B. Their performance is well-predicted by the scaling curve.

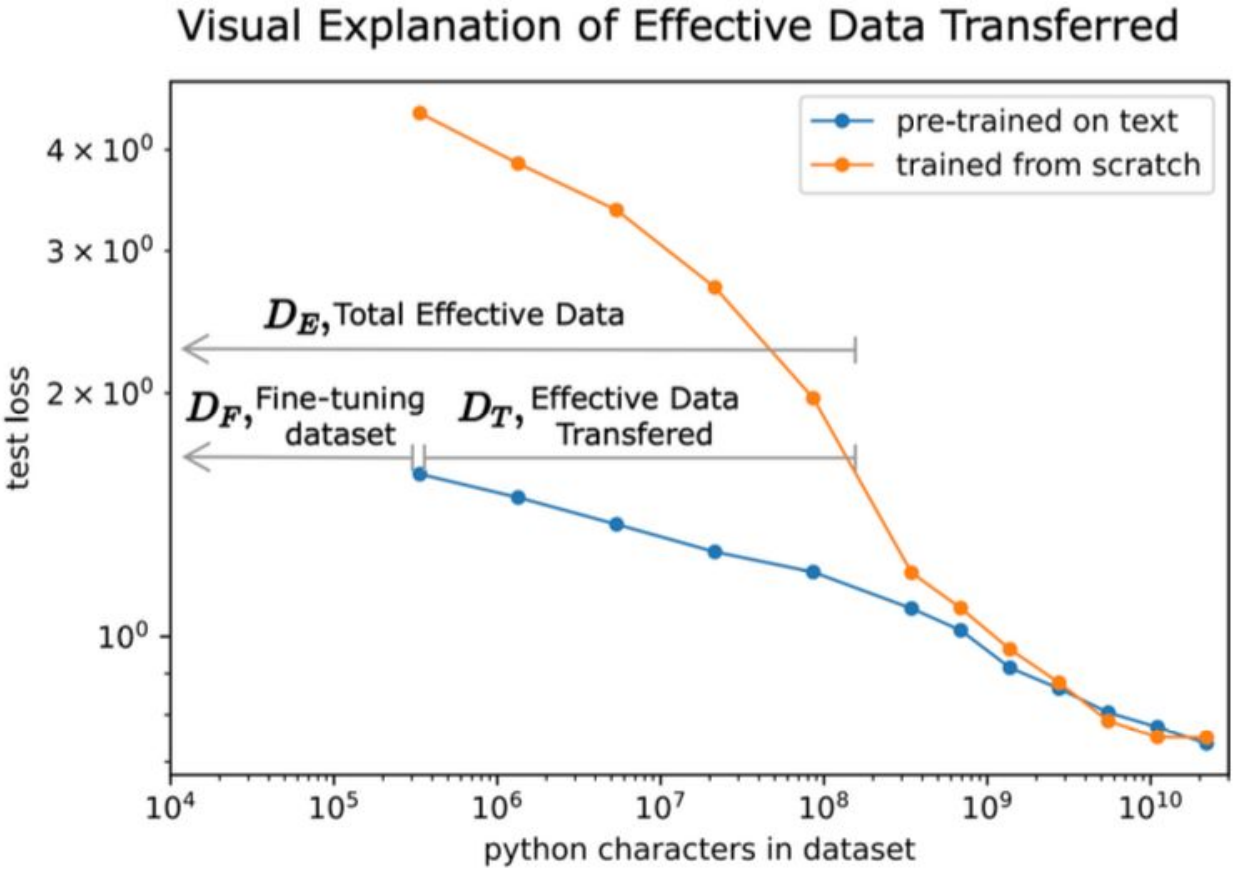
What if we run out of data?

- **Data-constrained setting**: we might want to train on much more than 2 trillion tokens but some programming languages have less tokens
- Option 1: repeat the data
- Option 2: mix in other data
- Option 3: transfer

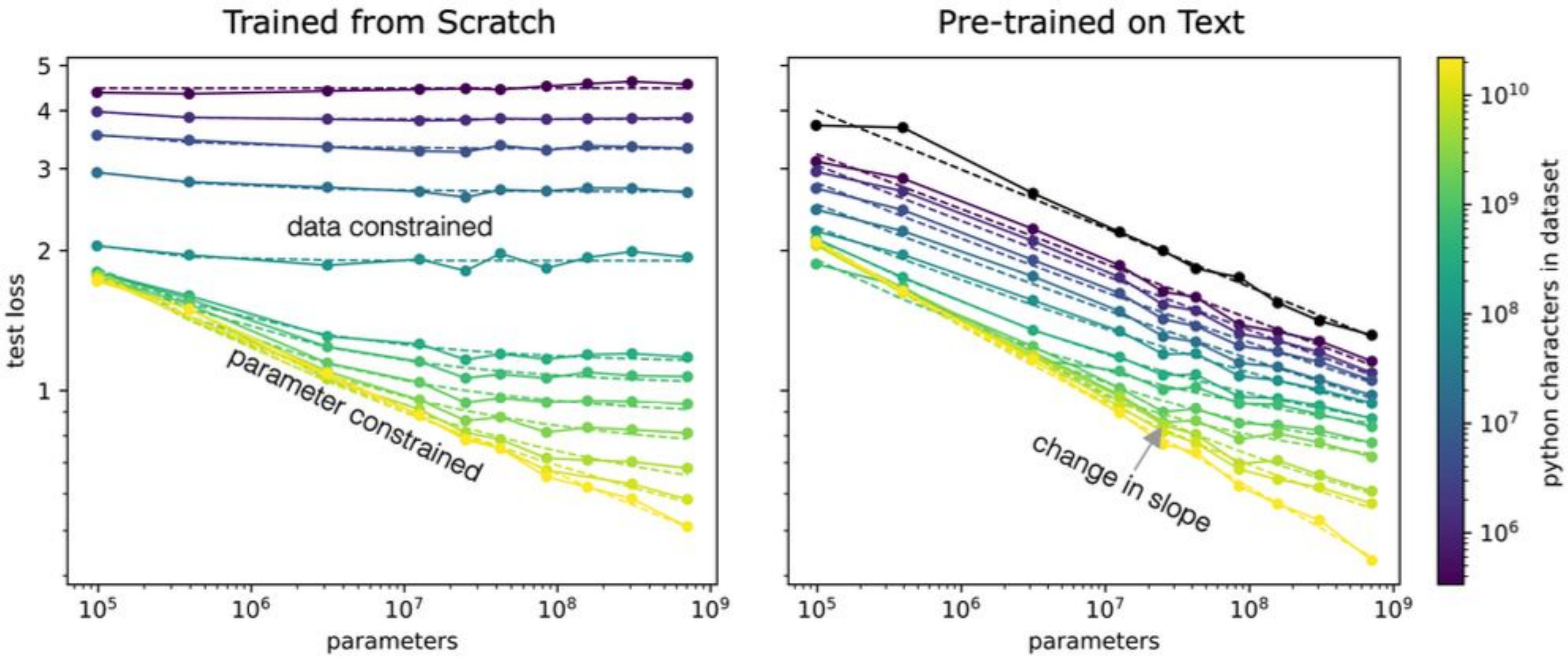




Scaling laws of transfer



Scaling laws of transfer



Low-data setting: without pretraining on text, we get no benefit from increasing parameters.

Scaling laws of transfer

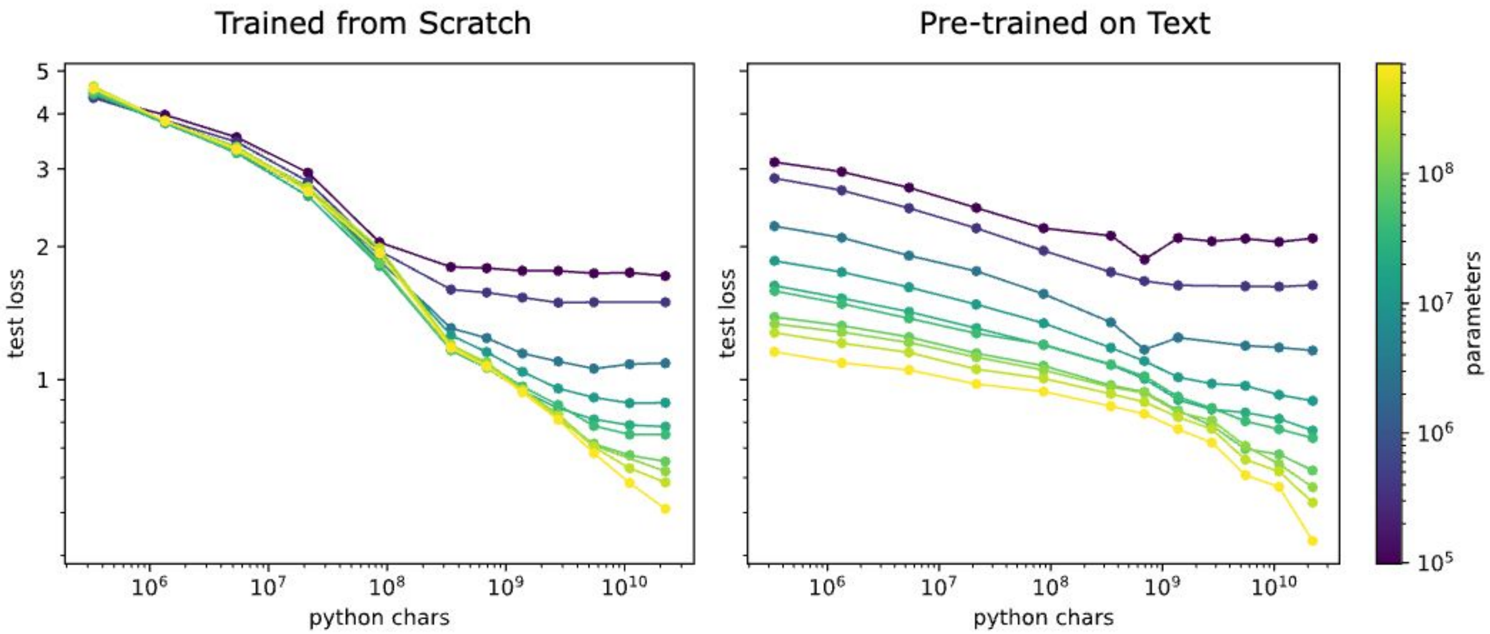


Figure 5 In the high data regime (purple lines), pre-training can effectively reduce the training set size. In other words, we get better performance training the 1M parameter models (purple) with our larger size datasets (>1e8) from-scratch.

Pretraining can ossify the model weights so that they don't adapt as well to the fine-tuning distribution in the high data regime

Formal and informal mathematics: Llemma

Problem (MATH Number theory 185): When a number is divided by 5, the remainder is 3. What is the remainder when twice the number is divided by 5? Show that it is 1.

Human-written informal proof: If our number is n, then $n \equiv 3 \pmod 5$. This tells us that

$$2n = n + n \equiv 3 + 3 \equiv 1 \pmod 5.$$

The remainder is 1 when the number is divided by 5.

Informal-to-formal (Isabelle):
{Problem, human-written informal proof}

```
theorem mathd_numbertheory_185:
  fixes n ::nat
  assumes "n mod 5 = 3"
  shows "(2 * n) mod 5 = 1"
```

```
proof -
  have "2 * n = n + n"  <ATP>
  also have "... mod 5 =
    (n mod 5 + n mod 5) mod 5" <ATP>
  also have "... = (3 + 3) mod 5"
    using assms <ATP>
  also have "... = 1" <ATP>
  finally show ?thesis <ATP>
qed
```

Formal-to-formal (Lean 4):

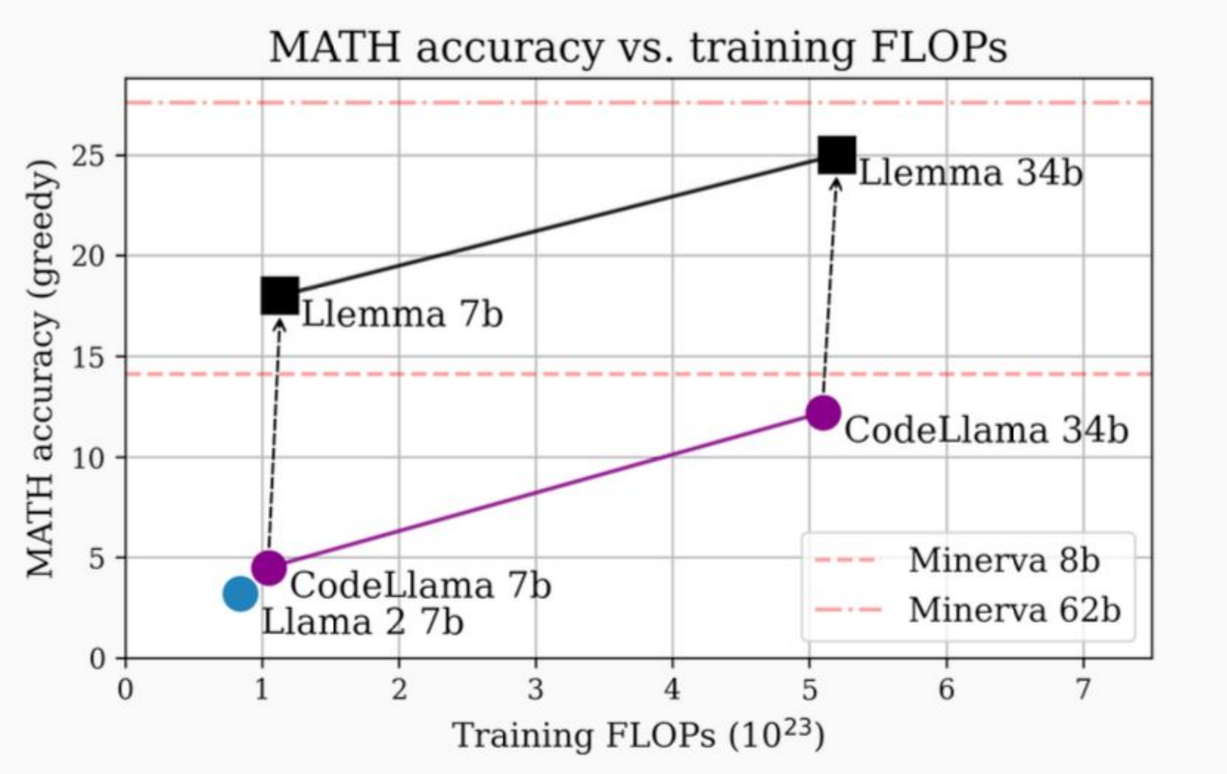
```
theorem mathd_numbertheory_185
  (n : ℕ) (h₀ : n % 5 = 3)
  : 2 * n % 5 = 1 := by

-- INPUT (step 1):
--   n: ℕ
--   h₀: n % 5 = 3
--   ⊢ 2 * n % 5 = 1
rw [mul_mod, h₀]

-- INPUT (step 2):
--   n: ℕ
--   h₀: n % 5 = 3
--   ⊢ 2 % 5 * 3 % 5 = 1
simp only [h₀, mul_one]
```

Figure 4: Example formal proofs from LLEMMA-7b. *Left:* The model is given a problem, informal proof, and formal statement, following Jiang et al. (2023). It generates a formal proof (starting with `proof -`) containing Isabelle code and calls to automation (shown as <*ATP*>). *Right:* The model is given a proof state, visualized as a grey comment, and generates the subsequent step (e.g. `rw [..]`).

Llemma: An Open Language Model For Mathematics
Azerbayev et al. ICLR (2024)

Data-constrained scaling: Llemma

Lemma:
- pretrain on code and web
- transfer by training on mathematical code, mathematical web data, scientific papers



Summary

- Pretraining fits the distribution of pretraining data
- Scaling laws let us forecast performance, allocate compute, and choose hyperparameters
- In low-data settings: repeat data, mix in other data, transfer

Not covered by scaling laws:

- Data quality: better data is more compute efficient
- Training objective: next-token may not be optimally efficient
- Distribution mismatch: example

Warning: calculating flops and param counts is an art!

```python
def chinchilla_params(seq_len, vocab_size, d_model, num_heads, num_layers, ffw_size):
    """ Parameters in the Chinchilla models. Unlike GPT they use relative positional embeddings. """
    # token embeddings only
    embeddings = d_model * vocab_size
    # transformer blocks
    attention = 3*d_model**2 + 3*d_model # weights and biases
    relative_pos = d_model**2 + 2*d_model # relative keys, content bias, rela
    attproj = d_model**2 + d_model
    ffw = d_model*ffw_size + ffw_size
    ffwproj = ffw_size*d_model + d_model
    layernorms = 2*2*d_model
    # dense
    ln_f = 2*d_model
    dense = d_model*vocab_size # note: no bias here
    # note: embeddings are not included in the param count!
    total_params = num_layers*(attention + relative_pos + attproj + ffw + ffw
    return total_params
```

```python
def chinchilla_flops(seq_len, vocab_size, d_model, num_heads, num_layers, ffw_size):
    """
    Calculate total number of FLOPs, see Chinchilla
    paper Appendix F as reference: https://arxiv.org/pdf/2203.15556.pdf
    """
    key_size = d_model // num_heads

    # embeddings
    embeddings = 2 * seq_len * vocab_size * d_model

    # attention
    # key, query, value projections
    attention = 2 * 3 * seq_len * d_model * (key_size * num_heads)
    # key @ query logits
    attlogits = 2 * seq_len * seq_len * (key_size * num_heads)
    # softmax
    attsoftmax = 3 * num_heads * seq_len * seq_len # 3* is for subtract (max), exp, divide (?)
    # softmax @ value reductions
    attvalue = 2 * seq_len * seq_len * (key_size * num_heads)
    # final linear
    attlinear = 2 * seq_len * (key_size * num_heads) * d_model
    att = attention + attlogits + attsoftmax + attvalue + attlinear
    # feed forward
    dense = 2 * seq_len * (d_model * ffw_size + d_model * ffw_size)

    # logits
    logits = 2 * seq_len * d_model * vocab_size

    # this is what you'd expect:
    # forward_flops = embeddings + num_layers * (att + dense) + logits
    # but:
    # per author correspondence apparently there is typo in the paper,
    # they do not count embeddings and logits to repro table 4. So instead:
    forward_flops = num_layers * (att + dense)
    backward_flops = 2 * forward_flops # as in Kaplan et al. 2020
    total_flops = forward_flops + backward_flops

    return total_flops
```

Source: https://github.com/karpathy/nanoGPT/blob/master/scaling_laws.ipynb