Lecture 9: Scaling Law 1

Marshall Meng and Dylan Fang

Stanford CS336 — Spring 2025

1 Background and Intuition

1.1 What Are Scaling Laws?

Scaling laws describe how the performance of a model changes when we scale its resources, such as

- the **amount of data** used for training,
- the **size of the model** (number of parameters),
- or the **training compute**.

When the model's loss or error is plotted against these quantities on a **log–log** plot, the curve often forms a straight line. This means the relationship follows a *power law*:

$$\text{Loss} = A \times n^{-\alpha} + C,$$

where n is the data or model size, α is the slope of the line, and A, C are constants. This straight-line behavior allows researchers to fit a scaling curve on small models and **predict how larger models will perform**—the "small → big" extrapolation idea.

1.2 Why Are Scaling Laws Important?

The lecture begins by asking practical scaling questions:

> "Should we make the model wider, deeper, or train longer?"

Instead of trial and error, scaling laws let us *measure small-scale trends* and forecast the performance of larger models. Because training large models costs enormous compute, this predictive ability saves both **time** and **resources**.

1.3 Historical Background

- **Banko & Brill (2001)** — Early NLP experiments showed that model accuracy continued improving steadily with more training data, hinting that scale itself drives progress.
- **Hestness et al. (2017)** — Provided the first quantitative evidence of power-law behavior in neural networks: test loss vs. dataset size formed nearly straight lines on log–log plots across vision and language tasks.
- **2020s onward** — Later studies extended these ideas to *joint* scaling laws combining data, model, and compute, forming the basis for scaling strategies used in GPT–3 and Chinchilla.

1.4 Toy Example and Intuition

Consider estimating the mean of a random variable. If we take n samples, the estimation error typically scales as

$$\text{Error} \propto \frac{1}{n}.$$

On a log–log plot, this appears as a straight line with slope -1. In higher-dimensional problems, the slope becomes shallower (for example, $-1/d$), illustrating that more complex tasks require more data to achieve the same accuracy. This provides simple intuition for the power-law patterns seen in real neural networks.

2 Data scaling details

2.1 Data Scaling Laws: Intuition and Foundations

Definition. **Data scaling laws** provide a simple mathematical rule that maps the size of a training dataset n to the model's performance (error or loss). Formally, we expect a relationship of the form

$$E(n) = A\, n^{-\alpha} + C,$$

where A and C are constants, and α is the *scaling exponent*. This tells us how quickly the model improves as we increase the amount of data.

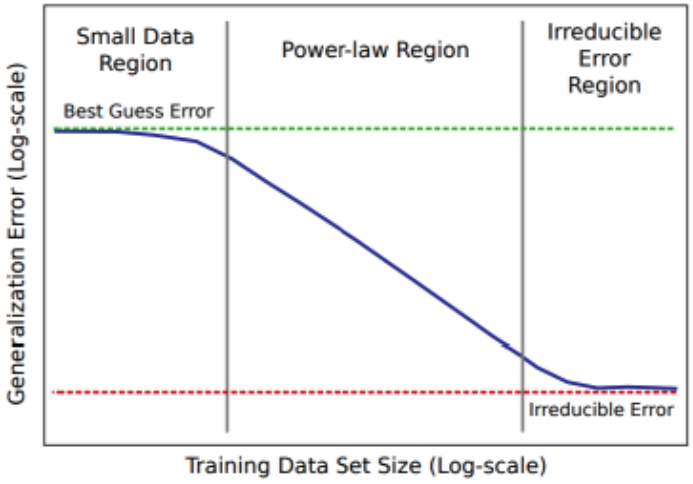


Figure 1: Three Regimes of Data Scaling. Error decreases monotonically as dataset size increases, passing through three characteristic regions.

Three characteristic regions.

- **Small Data Region:** Too little data for learning; model memorizes or guesses (error nearly constant).
- **Power-law Region:** Predictable improvement following $E(n) \propto n^{-\alpha}$.
- **Irreducible Error Region:** Further data no longer helps; performance hits a noise floor.

Overall, the curve is **monotonic and logistic-like**: flat $\rightarrow$ steep $\rightarrow$ flat.

2.1.1 Empirical Observation: Language Models

In large-scale language models, the same power-law pattern appears when plotting test loss against dataset size on a log–log plot.

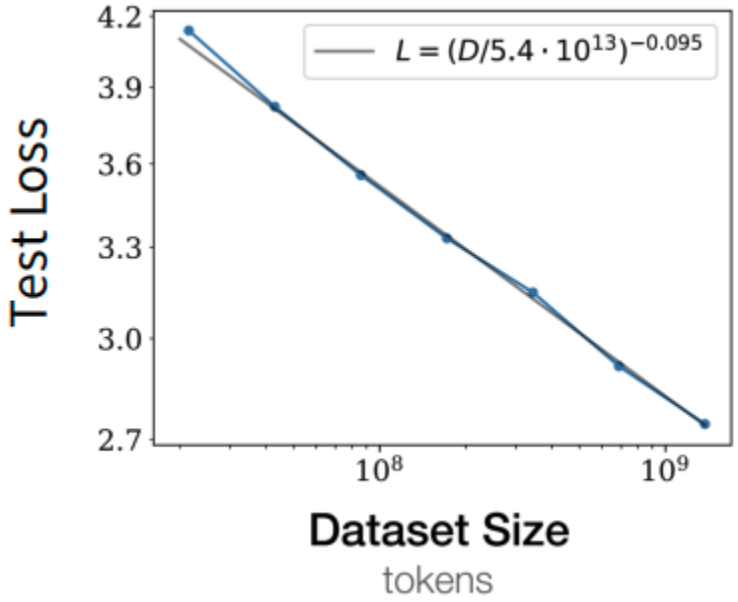


Figure 2: Empirical data scaling for language models. Test loss decreases linearly with log-dataset size, following a power law.

This "scale-free" or "power-law" behavior means that increasing the dataset size by a constant factor (e.g., 10×) produces a predictable, consistent reduction in loss. For example, Kaplan et al. (2020) observed

$$L = \left(\frac{D}{5.4 \times 10^{13}} \right)^{-0.095},$$

where L is the test loss and D the number of training tokens.

Intuition. Scaling laws arise because **estimation error decays polynomially** with the number of samples. For many simple estimation problems, the error follows

$$E(n) \propto \frac{1}{n^{\alpha}},$$

for some constant $\alpha > 0$. This natural polynomial decay explains why the log–log curve of error versus data size becomes linear: it is the direct mathematical signature of a power law.

Key insight.

- The curve must be *monotonic*: more data never hurts.
- In the middle (power-law) region, the behavior is smooth and predictable.
- This predictability allows us to extrapolate model performance from small-scale experiments.

2.2 Scaling Law Exponents Across Tasks

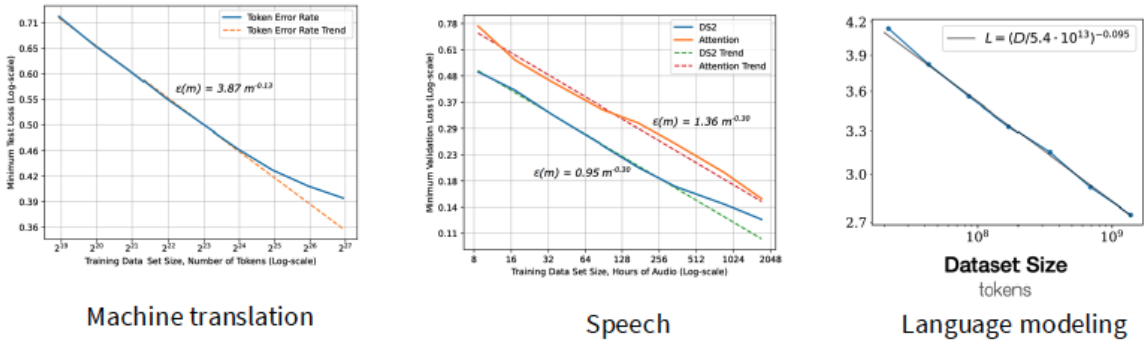


Figure 3: Empirical neural scaling laws across tasks. Each panel shows a power-law fit for test error vs. dataset size on log–log axes. The fitted forms (e.g., $\varepsilon(m) = A\,m^{-\alpha}$ or $L = (D/D_0)^{-\alpha}$) yield **shallow exponents** ($\alpha \approx 0.095$–0.30), much smaller than the classical $1/n$ expectation ($\alpha = 1$).

Interpretation. Let $\varepsilon(m)$ denote the **generalization error** (or test loss) obtained after training a model on a dataset of size m. It measures how well the trained model performs on unseen data—a lower value indicates better generalization. In practice:

- The **solid line** on each plot represents measured test errors at various dataset sizes.
- The **dashed line** represents the fitted power-law trend $\varepsilon(m) = A\,m^{-\alpha} + C$.

Meaning of each term.

- m: the amount of training data (e.g., number of tokens, examples, or hours of audio). Some papers denote this by n; both mean "dataset size."
- $\varepsilon(m)$: the generalization error achieved at that dataset size. It is the quantity plotted on the y-axis.
- A: a constant scaling factor that vertically shifts the curve. It depends on the task, model family, and data quality.
- α: the **scaling exponent** (positive value). On a log–log plot, the actual slope of the line is $-\alpha$. Larger α means error decreases *faster* with data; smaller α means slower, shallower improvement.
- C: the irreducible error floor (noise or ambiguity in the task). Often omitted when fitting the mid-range "power-law" region.

Observed behavior. Across machine translation, speech, and language modeling, we find

$$\varepsilon(m) = A\,m^{-\alpha} \quad \text{with} \quad \alpha \in [0.095,\, 0.30] \text{ (approx.)}.$$

On log–log axes, these curves appear linear, confirming a power-law relation, but the exponents are far smaller than the classical $1/n$ scaling expectation ($\alpha = 1$).

Interpreting the exponent α. The exponent α governs how quickly the model benefits from more data.

- **Classical theory:** For simple parametric models (e.g., linear regression), error $\propto 1/n \Rightarrow$ slope -1.
- **Neural networks:** Empirically, $\alpha \approx 0.1$–$0.3 \Rightarrow$ slopes -0.1 to -0.3. This means performance improves predictably but more slowly: each $10\times$ data increase yields smaller relative gains.

For instance, if $\alpha = 0.1$, then

$$\frac{\varepsilon(10m)}{\varepsilon(m)} = 10^{-0.1} \approx 0.79,$$

so ten times more data reduces error by only $\sim 21\%$. In contrast, the ideal $1/n$ case ($\alpha = 1$) would yield a $10\times$ reduction.

Relation to task and model difficulty. The exponent α captures how *data-efficient* learning is for a given task and model:

- **Larger** α (≈ 0.3 in speech) $\Rightarrow$ easier or more data-efficient: the model learns faster per added data.
- **Smaller** α (≈ 0.1 in LMs) $\Rightarrow$ harder or less data-efficient: diminishing returns appear sooner.
- Differences in α arise from task complexity, input dimensionality, model architecture, and data quality.

2.3 Intrinsic Dimensionality Theory of Data Scaling Laws

The Core Question (Bahri et al. 2021) We observe that model error improves with data size (n) following a power law: Error $\propto 1/n^{\alpha}$. But *why* does this happen? And what determines the value of α (the scaling exponent)? This theory proposes that α is directly linked to the data's *true complexity*, known as its **intrinsic dimensionality**.

Key Concepts Explained

- **Scaling Exponent (α):** This is the **slope** of the `log(Error)` vs. `log(Data)` line. It's a single number that measures *how fast* the model learns from new data.
  - **Large α:** Fast learning (error drops quickly).
  - **Small α:** Slow learning (error drops slowly).
- **Intrinsic Dimensionality (d_I):** This is the *true number of variables* or "degrees of freedom" in your data. It's almost always much lower than the *apparent* dimension (e.g., the total number of pixels).
  - **Analogy:** Imagine a 1-megapixel photo (1,000,000 pixels) of a clock face. The *apparent* dimension is 1,000,000. But if the dataset only shows the clock at different times, the *intrinsic* dimension is just $d_I = 1$, because the only thing that truly varies is the "time".
  - **Example:** MNIST (digits) is simple, so its d_I is low (estimated $\sim$10-15). CIFAR-100 (100 object types) is much more complex, so its d_I is higher.

The Central Hypothesis The data's true complexity (d_I) determines its learning speed (α). The logic is:

- **Simple Data (low d_I)** $\rightarrow$ Easy to learn $\rightarrow$ Error drops fast $\rightarrow$ **Large** α
- **Complex Data (high d_I)** $\rightarrow$ Hard to learn $\rightarrow$ Error drops slowly $\rightarrow$ **Small** α

This implies an inverse relationship: the faster the learning (larger α), the simpler the data (smaller d_I). Or, $\alpha \propto 1/d_I$.

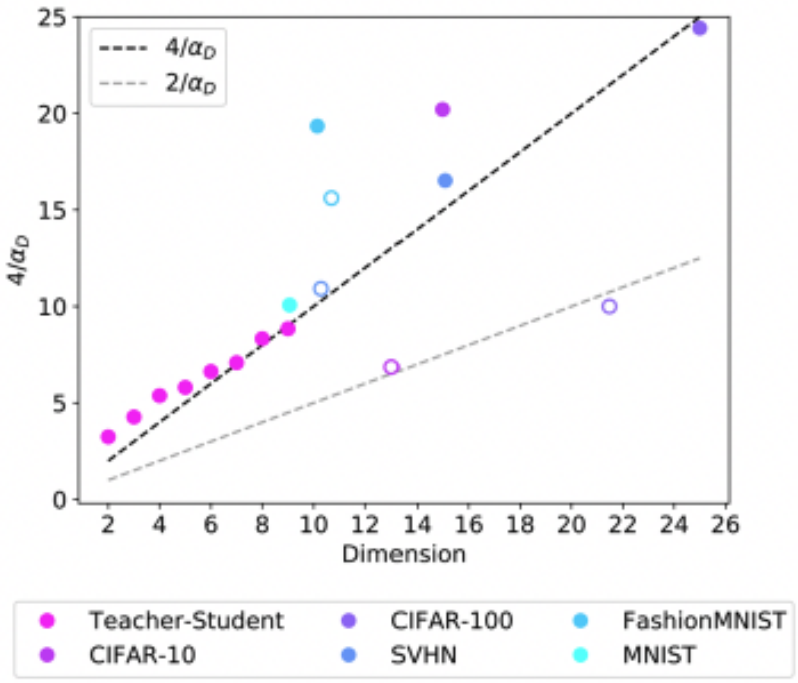


Figure 4: Empirical test of the theory. The graph plots the measured complexity (d_I) vs. the measured learning speed (calculated as $4/\alpha_D$).

Interpreting the Graph (The Experiment) This graph is the experimental test of the hypothesis $d_I \propto 1/\alpha$. It plots the estimated complexity of a dataset against the measured learning speed on that dataset.

- **x-axis (Dimension):** The **estimated intrinsic dimension (d_I)**. This is the data's "true complexity." Datasets on the left (like MNIST) are simple; datasets on the right (like CIFAR-10) are more complex.
- **y-axis ($4/\alpha_D$):** A value calculated from the **measured scaling exponent (α_D)**. The exponent α_D is the slope of the `log(Error)` vs. `log(Data)` curve for that dataset.
  - Because α_D is in the denominator, a *small* exponent (slow learning) results in a *high* Y-value.
  - This axis is chosen specifically to test the theory $d_I = 4/\alpha_D$. If this theory is true, a dataset with $d_I = 8$ (x-value) should have a measured $4/\alpha_D = 8$ (y-value), and the dot should fall on the perfect $Y = X$ diagonal line.

- **Dashed Lines (The Theories):** These are the theoretical predictions.
  - $4/\alpha_D$ **line:** This is the main theory being tested ($d_I = 4/\alpha_D$). It is the $Y = X$ line.
  - $2/\alpha_D$ **line:** This is an alternative, weaker theory ($d_I = 2/\alpha_D$).
  - The data points (dots) fall much closer to the $4/\alpha_D$ line, which strongly supports that specific theory.
- **The Dots (The Data):** Each dot represents one experiment on one dataset.
  - **Color** identifies the dataset (e.g., pink is CIFAR-10, light blue is MNIST).
  - **Multiple dots of the same color** (e.g., several pink dots) likely represent different runs using different algorithms to *estimate* the intrinsic dimension (the x-value), which is a "sketchy" or noisy process.
- **Overall Finding:** The graph shows a clear bottom-left to top-right trend. This visually confirms the hypothesis: datasets with higher intrinsic dimension (more complex, further right on x-axis) exhibit smaller scaling exponents (slower learning, higher on y-axis).

Conclusion This theory argues that a dataset's **scaling exponent (α)** is inversely related to its **intrinsic dimension (d_I)**. This provides a direct link between the data's complexity and its learning rate:

- **Simple Data (Low d_I):** Datasets with a low intrinsic dimension (like MNIST) are "easier" to learn. They exhibit a **large scaling exponent (α)**, meaning their test loss drops *quickly* as data size increases.
- **Complex Data (High d_I):** Datasets with a high intrinsic dimension (like CIFAR-100) are "harder" to learn. They exhibit a **small scaling exponent (α)**, meaning their test loss drops *slowly*, requiring much more data for the same performance gain.

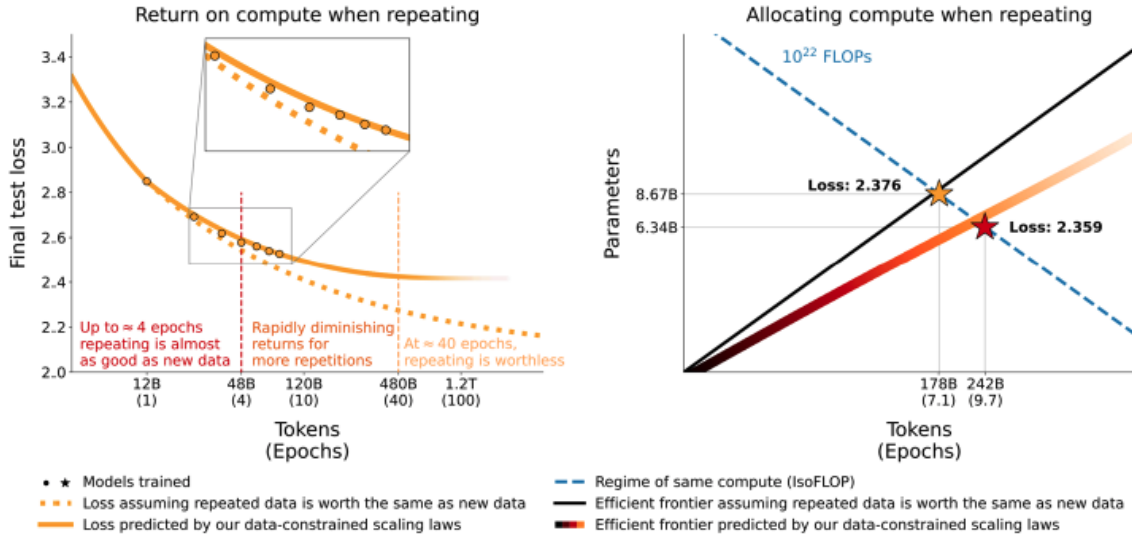
2.4 Advanced Data Scaling: Data Composition and Distribution Shift

Motivation Thus far, scaling laws have addressed how dataset *size* relates to performance. A more advanced, related question is: how does dataset *composition* (the mixture of different data sources) affect performance? This line of inquiry helps answer practical questions, such as:

- Picking an optimal data mixture using small-scale models.
- Deciding whether it is better to repeat existing data or collect new data.
- Balancing data quality with the repetition rate.

2.5 Scaling Laws Under Data Repetition

Motivation In practice, datasets are finite. We must eventually repeat data (i.e., train for multiple epochs). This section explores how data repetition affects model performance and optimal training strategy.



Interpreting the "Return on compute" Graph (Left) This graph shows the diminishing value of repeating data within a 12B tokens dataset. The (Epochs) number shows how many times that 12B dataset was repeated.

- It compares two scaling laws: an **ideal** one (dotted line) , which assumes repeated data is as valuable as new data, and a **realistic** one (solid line), which models the actual, observed performance.
- **Key Insight 1:** For a few repetitions (up to ≈ 4 epochs), the realistic performance is very close to the ideal one. Repeating data is highly effective at first.
- **Key Insight 2:** After ≈ 4 epochs, the model shows "rapidly diminishing returns". The realistic line flattens out, diverging from the ideal line.
- **Key Insight 3:** At very high repetition rates (≈ 40 epochs), the realistic line becomes flat, meaning more training provides no benefit and "repeating is worthless".

Interpreting the "Allocating compute" Graph (Right) This graph shows how diminishing returns affect the optimal training strategy for a **fixed compute budget** (an "IsoFLOP" line, shown in dashed blue).

- The fixed budget (dashed blue line) presents a tradeoff: for the *same cost*, you can train a large model on few tokens (top-left) or a small model on many tokens (bottom-right).
- **Ideal Optimum (Yellow Star):** This point lies on the **ideal efficient frontier (black line)**, which wrongly assumes repeated data is as good as new. Following this, the best strategy would be an 8.67B model trained for 7.1 epochs.
- **Realistic Optimum (Red Star):** This point lies on the **realistic efficient frontier (orange line)**, which correctly models diminishing returns. This frontier shows the true optimal strategy for this budget is a **smaller model (6.34B parameters)** trained for **more epochs (9.7)**, as it achieves a better final loss.

Interpreting the Equation The realistic, data-constrained scaling law is modeled by the following equation, which calculates the **"Effective data" (D')**—the true, "discounted" learning value the model gets from training.

$$D' = U_D + U_D R_D^* (1 - e^{\frac{-R_D}{R_D^*}})$$

- **D':** The "Effective data" – the true learning value the model gets, which actually predicts the loss.
- **U_D:** The number of "Unique tokens" in the dataset (e.g., 12B). This is the value of the "fresh meal" or the first epoch.
- **R_D:** The number of "Repetitions" – i.e., the number of epochs *after the first one.* (e.g., at 4 total epochs, $R_D = 3$).
- **R_D^*:** A "Constant" that controls how fast the value of repetitions decays.
- **Intuition:** The total effective data is the value of the first epoch (U_D) plus a "bonus" from all repetitions. The '$(1 - e^{-x})$' term is a "braking" function. As repetitions (R_D) get large, this term approaches 1, meaning the "bonus" value hits a hard maximum "ceiling" of $U_D R_D^*$. This mathematically models the diminishing returns seen in the graphs.

Concrete Example Let's use the graph's numbers: $U_D = 12$B tokens. We'll assume the constant $R_D^* = 4$. The formula becomes: $D' = 12\text{B} + (12\text{B} \times 4) \times (1 - e^{-R_D/4}) = 12\text{B} + 48\text{B} \times (1 - e^{-R_D/4})$.

- **Epoch 1 ($R_D = 0$):**
  - **Total Processed:** 12B
  - **Effective Data (D'):** $12\text{B} + 48\text{B} \times (1 - e^0) = 12\text{B} + 0 = \mathbf{12B}$.
  - **Efficiency (D' / Processed):** $12\text{B}/12\text{B} = \mathbf{100\%}$.
- **Epoch 4 ($R_D = 3$):**
  - **Total Processed:** $12\text{B} \times 4 = 48\text{B}$
  - **Effective Data (D'):** $12\text{B} + 48\text{B} \times (1 - e^{-3/4}) \approx 12\text{B} + 48\text{B} \times (0.528) = 12\text{B} + 25.3\text{B} = \mathbf{37.3B}$.
  - **Efficiency (D' / Processed):** $37.3\text{B}/48\text{B} \approx \mathbf{78\%}$.
- **Epoch 40 ($R_D = 39$):**
  - **Total Processed:** $12\text{B} \times 40 = 480\text{B}$
  - **Effective Data (D'):** $12\text{B} + 48\text{B} \times (1 - e^{-39/4}) \approx 12\text{B} + 48\text{B} \times (1) = \mathbf{60B}$.
  - **Efficiency (D' / Processed):** $60\text{B}/480\text{B} = \mathbf{12.5\%}$.

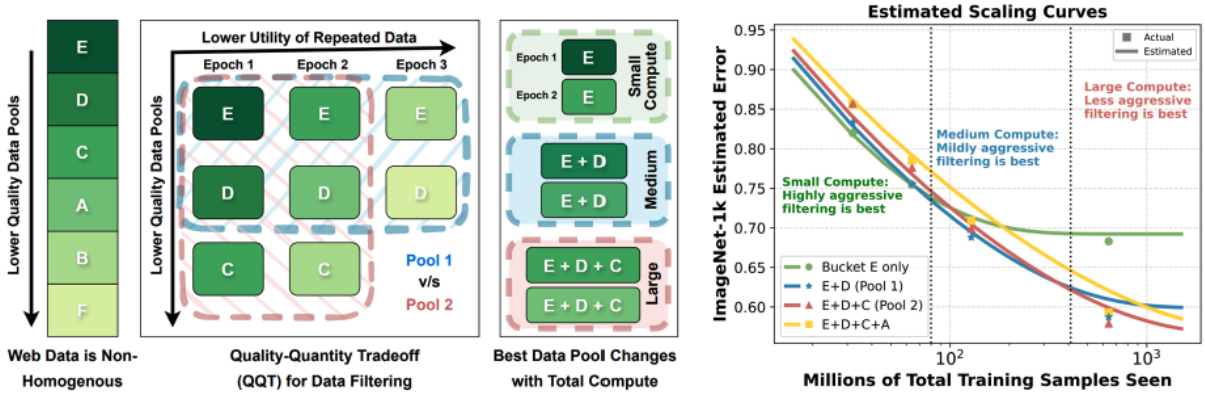
This example confirms the main insight: as repetitions (R_D) grow, the total effective data (D') increases but hits a ceiling (60B), while the *proportion* or *efficiency* of each token processed drops dramatically.

2.6 Data Selection Scaling and Finiteness

Motivation Given that repeated data (epochs) has diminishing returns, and real-world data is "non-homogenous" (i.e., of varying quality), the optimal data selection strategy must be **adaptive to scale**.

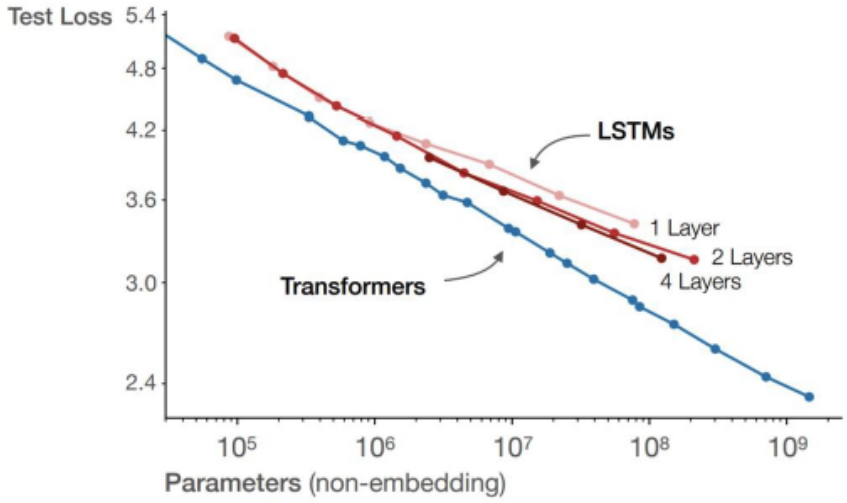
The Core Tradeoff (QQT) This creates a "Quality-Quantity Tradeoff" (QQT) for data filtering:

- **Highly Aggressive Filtering (e.g., "Bucket E only"):** This yields a **high-quality** but **small-quantity** dataset. It will be repeated many times, hitting the diminishing returns wall very quickly.
- **Less Aggressive Filtering (e.g., "E+D+C"):** This yields a **larger-quantity** dataset, but the **average quality is lower**. It will be repeated fewer times, hitting the diminishing returns wall much later.



Interpreting the "Estimated Scaling Curves" Graph This graph provides the experimental proof for how the optimal strategy changes with compute.

- **Axes:** The graph plots Model Error (Y-axis, lower is better) against Total Compute/Samples Seen (X-axis).
- **Lines:** Each line represents a different data pool, from highly aggressive (green, "E only") to less aggressive (red, "E+D+C").
- **The Crossover Finding:** The best-performing line changes as compute increases.
  1. **Small Compute (Left):** The **Green line ("E only")** is best. With a small budget, using only the highest-quality data is the most efficient strategy.
  2. **Medium Compute (Middle):** The Green line flattens as it hits the repetition wall. The **Blue line ("E+D")** takes over as the optimal choice.
  3. **Large Compute (Right):** The Blue line also flattens. The **Orange line ("E+D+C")** becomes the best. This proves that with a large compute budget, avoiding repetition by including more data (even of lower quality) is the only way to continue improving.



Conclusion The optimal data-filtering strategy is not fixed. It must become *less aggressive* as the total training compute increases.

3 Model/optimization scaling

Model scaling is the practice of using scaling laws to make efficient design and resource allocation decisions when building very large language models. The core idea is to find a simple, predictive procedure to answer critical questions without having to expensively train many huge models.

3.1 Model Engineering: Architecture

Transformers vs. LSTMs (Kaplan+) This analysis uses scaling laws to efficiently answer: "Are Transformers better than LSTMs?"

- **The Old Method:** The "brute force" approach would be to train a massive, GPT-3-sized LSTM, which is prohibitively expensive.
- **The Scaling Law Method:** Instead, train many small models of both types and compare their scaling curves.
- **Graph Interpretation:** The plot shows Test Loss (Y-axis) vs. Model Parameters (X-axis).
- **Finding:** The scaling curve for Transformers is consistently below the curve for LSTMs. This proves that for any given parameter count, Transformers are more efficient and achieve a lower loss. They "scale" better.

Comparing Many Architectures (Tay et al.) This work expands the comparison to many different modern architectures (e.g., ALBERT, Switch Transformer, MLP Mixer).

- **Graph Interpretation:** The main plot shows performance (Y-axis: Negative Log-Perplexity, higher is better) against compute cost (X-axis: FLOPS).

- **Finding:** Most of these advanced Transformer variants cluster along a similar "efficient frontier". This suggests that while the jump from LSTMs to Transformers was a massive improvement, the differences *between* modern Transformer-like architectures are less critical, as most scale in a similarly predictable way.

3.2 Optimizer Choice

Adam vs. SGD (Hestness+ 2017) This section addresses the choice between optimizers, specifically comparing Adam and SGD, using Recurrent Highway Networks (RHNs) on varying dataset sizes. (Note: This research predates the widespread adoption of Transformers).

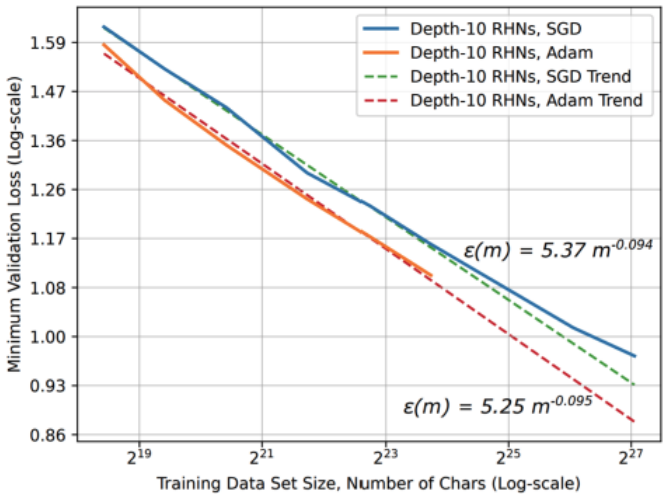


Figure 5: Scaling behavior of Adam vs. SGD optimizers with increasing dataset size.

- **Graph Axes:** The x-axis represents Training Data Set Size (Number of Chars) on a log-scale, and the y-axis shows Minimum Validation Loss (log-scale), where lower is better.
- **Solid Lines:** Show the empirical performance of Depth-10 RHN models using SGD (blue) and Adam (orange).
- **Dashed Lines (Trends):**
  - **SGD Trend (Green):** The extrapolated scaling law for SGD, predicting its performance at larger data scales.
  - **Adam Trend (Red):** The extrapolated scaling law for Adam, predicting its performance at larger data scales.
- **Key Finding - Crossover Effect:**
  - **Small Data Scale:** Adam (orange line) initially achieves lower loss than SGD (blue line), indicating faster and better performance for smaller datasets.
  - **Large Data Scale:** As data size increases, the trend lines cross. SGD (green dashed trend) is predicted to eventually achieve a lower validation loss than Adam (red dashed trend).

Conclusion While Adam can offer faster initial convergence and better performance on smaller datasets, SGD appears to scale more robustly and achieves superior performance when training with very large datasets. This highlights that the "best" optimizer depends on the scale of data being used.

3.3 Depth/Width: Number of Layers

Question Does a model's depth (number of layers) or width make a huge difference?

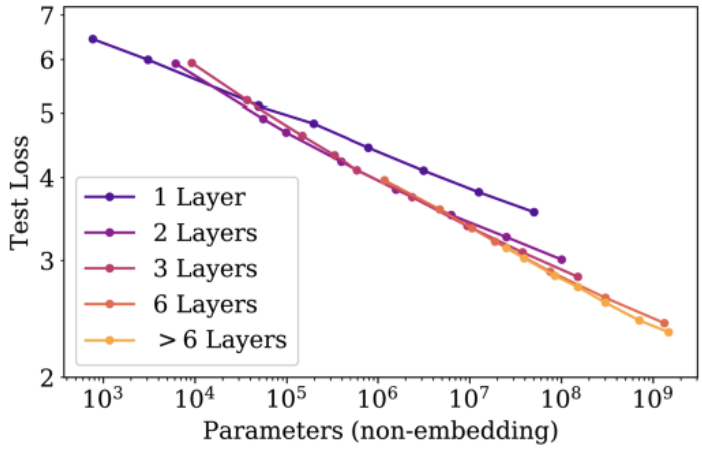


Figure 6: Scaling comparison for models with different layer counts.

Graph Interpretation The graph plots Test Loss (Y-axis) against the number of non-embedding parameters (X-axis) for models with different layer counts.

- The 1-layer model is significantly worse than all others.
- The 2-layer model is a major improvement over the 1-layer, but still performs worse than deeper models.
- The 3-layer, 6-layer, and 6+ layer models are all clustered tightly.

Conclusion: Adding layers provides a large benefit at first, but this effect shows rapidly diminishing returns. After ≈ 3 layers, adding more depth does not significantly improve performance for a fixed parameter count.

3.3.1 Depth/Width: Not All Parameters Are Equal

Concept A model's parameters can be split into two types, which do not contribute to performance equally:

- **Embedding Parameters:** A large, simple lookup table of size Vocabulary Size $\times$ d_{model}. Its job is only to convert token IDs (like "the") into a vector. It does not perform complex reasoning.
- **Non-Embedding (Computational) Parameters:** The "smart" parameters in the Transformer blocks (i.e., the weights in the Attention and FFN layers). This is where the model's actual computation and "thinking" happens.

Toy Example For a tiny **1-layer** model with:

- 1000 vocab tokens
- $d_{\text{model}} = 8$
- 500 computational parameters (in its single Transformer block)

The parameter counts would be:

- **Embedding Parameters:** $1000 \times 8 = 8,000$
- **Non-Embedding Parameters:** 500
- **Total ("with embedding") Parameters:** $8,000$ (dumb) $+ 500$ (smart) $= 8,500$

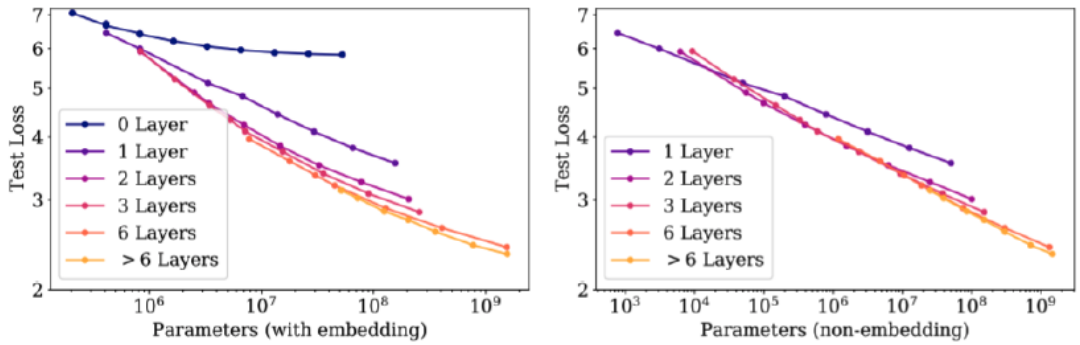


Figure 7: Comparison of scaling when plotting total parameters vs. only non-embedding parameters.

Graph Interpretation The two plots show the *exact same models*, but use a different calculation for the x-axis.

- **Left Plot ("with embedding"):** This graph is "polluted." It plots loss against the *total* parameter count. The "0 Layer" model (dark blue) proves this is a bad idea: it has 10^7 (10 million) parameters, but they are *all* "dumb" embedding parameters, so its loss is terrible. This large, fixed embedding cost skews the x-axis for all other models and hides the true scaling relationship.
- **Right Plot ("non-embedding"):** This graph is "clean." It plots loss against only the "smart" *computational* parameters.
  - The "0 Layer" model disappears, as it has 0 non-embedding parameters.
  - The other models (1 Layer, 2 Layers, etc.) now form clean, predictable, and parallel lines. This reveals the true scaling law.

Conclusion This slide proves that to get a clean and predictive scaling law, you **must** ignore the embedding layer and plot performance against the **non-embedding parameter count**.

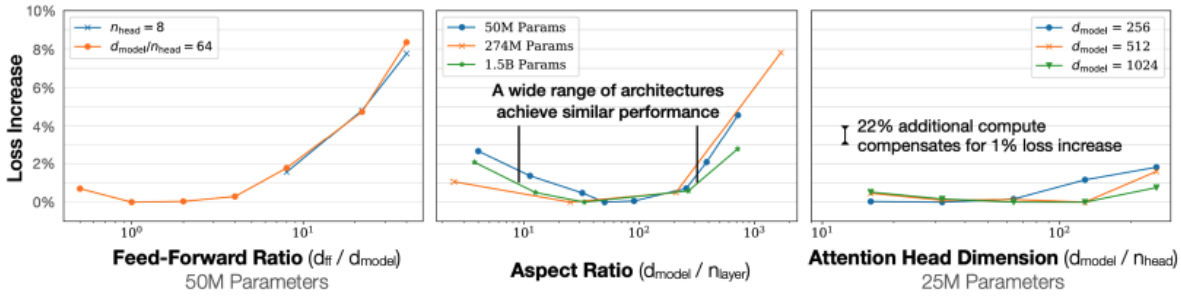


Figure 8: Impact of FFN Ratio, Aspect Ratio, and Head Dimension on loss.

3.3.2 Depth/Width: Other Transformer Hyperparameters

Question For a fixed number of non-embedding parameters, does the model's "shape" matter?
Main Finding: No. Performance depends "very mildly" on model shape.

Graph Interpretations All three plots show "Loss Increase" (a penalty) on the Y-axis.

- **Feed-Forward Ratio ($d_{\mathbf{ff}}/d_{\mathbf{model}}$):** This is the expansion factor in the FFN. The graph has a "U" shape, showing the optimal (lowest-loss) ratio is between 2 and 4. Moving away from this sweet spot increases the loss.
- **Aspect Ratio ($d_{\mathbf{model}}/n_{\mathbf{layer}}$):** This compares model width to depth. The "U" curve is extremely flat and wide, showing that the ratio can vary dramatically (from 10 to 400) with almost no loss penalty. This holds true for small (50M) and large (1.5B) models.
- **Attention Head Dimension ($d_{\mathbf{model}}/n_{\mathbf{head}}$):** This is simply d_{head}. This curve is also very flat, showing that any head dimension between ≈ 20 and ≈ 100 is optimal.

Conclusion: As long as you are within a very wide, reasonable range for these shape hyperparameters, the model's performance will be near-optimal.

3.4 Batch Size

Defining Critical Batch Size Batch size refers to the number of examples used in a single gradient update. The choice of batch size involves a trade-off between gradient accuracy and training speed.

- **Loss Landscape Plot (Left):** This illustrates the trade-off. A small batch (red) gives a "noisy" gradient, leading to an inefficient, zig-zag path to the minimum. A larger batch (blue) averages out this noise, giving a "cleaner" gradient and a more direct path.
- **Training Speed Plot (Right):** This shows the impact of batch size on wall-clock training speed.
  - **Perfect scaling:** For small batches, doubling the batch size doubles the training speed.
  - **Ineffective scaling:** For large batches, doubling the batch size provides no speedup and wastes compute.

- **Critical Batch Size:** This is the "knee" or "elbow" of the speed curve—the point where diminishing returns begin. It is the largest batch size that still benefits from the "perfect scaling" regime.

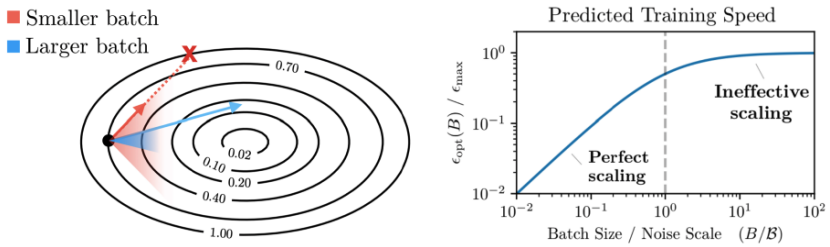


Figure 9: Plots of Critical Batch Size vs. Performance.

Critical Batch Size vs. Performance The optimal or "critical" batch size is not a single, fixed number.

- **Graph Interpretation:** This graph plots the target "Train Loss" (X-axis) against the measured "Critical Batch Size" (Y-axis). The critical batch size increases as the target loss decreases. This means at the start of training (high loss), a small batch is efficient. But to reach a very low loss at the end of training, you need a much larger batch to average out the noise and find the precise path to the minimum.
- **Key Finding:** As training progresses and the target loss gets *smaller* (moving right on the graph), the critical batch size gets *larger*. This means the optimal strategy is to increase the batch size as training proceeds.
- **Equation:** The formula $C_{\text{min}}(C) = C/(1 + B/B_{\text{crit}}(L))$ models this inefficiency.
  - B is your chosen batch size, and $B_{\text{crit}}(L)$ is the critical batch size for a given loss L.
  - When $B \ll B_{\text{crit}}$, the denominator is ≈ 1, and compute is used perfectly (the "minimum compute" state).
  - When $B \gg B_{\text{crit}}$, the denominator is large, and compute is used inefficiently.

Selecting the Optimal Batch This answers how to allocate a larger compute budget.

- **The Question:** Should we use huge batches for the same steps, or fixed batches for more steps?
- **Graph Interpretation:** The graph plots the total "Compute" budget (X-axis) against the number of training "Steps" (Y-axis).
  - **Orange Line (fixed-batch):** This strategy is inefficient. To spend more compute, the number of steps must explode.

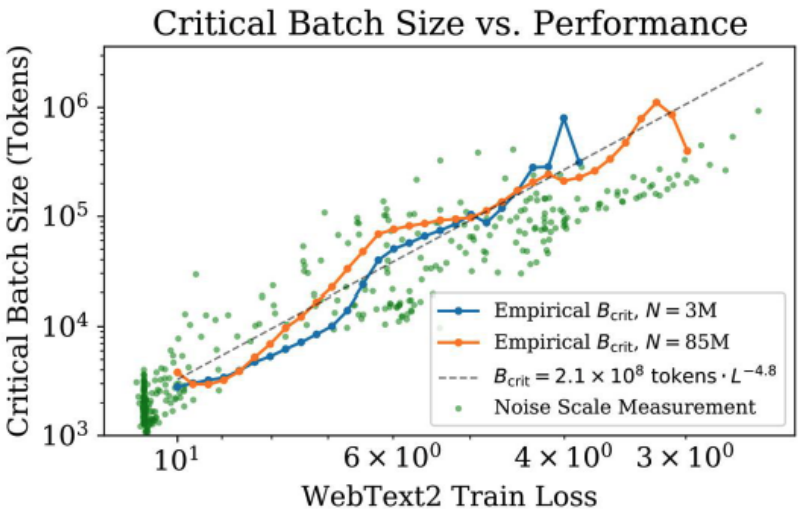


Figure 10: Selecting the optimal batch size for a given compute budget.

  - **Blue Line (adjusted):** This is the optimal strategy. As the compute budget increases, the number of steps stays nearly constant.

- **Conclusion:** The optimal way to scale training is to **use larger batch sizes**, not more steps. This is "good news for data parallel processing" as it fits the modern hardware paradigm of distributing a single large batch over many GPUs.

3.5 Learning Rates: µP and Scale-Aware LR Choices

Motivation A major challenge in scaling is that hyperparameters, especially the **learning rate (LR)**, do not transfer from small models to large ones. The learning rate is the "step size" the model uses to update its weights. An optimal LR is critical for good performance.

The Problem ("Standard Practice" Graph) The left graph shows what happens with "naive" scaling.

- **The Experiment:** This experiment tests a single learning rate (e.g., 'log(LR) = -12') and applies that *exact same* LR to all models, from Width 128 to 8192.

- **The Result:** The graph shows the optimal LR "shifts". The best LR for the small model (Width 128) is around -12, but the best LR for the large model (Width 8192) is around -14.

- **Conclusion:** This is a critical problem. The best LR found on a small, cheap model is *wrong* for a large, expensive model, which defeats the purpose of using scaling laws.

The Solution: µP ("Our Work" Graph) The right graph shows the power of µP, a "scale-aware" method.

- **The Experiment:** This "smart" experiment uses the µP recipe (from the table). The x-axis, `log(LearningRate)`, is now the **base_lr** (l). For each model, the *actual* LR used in training is scaled: Actual LR = base_lr$/r$ (where r is the scaling factor).

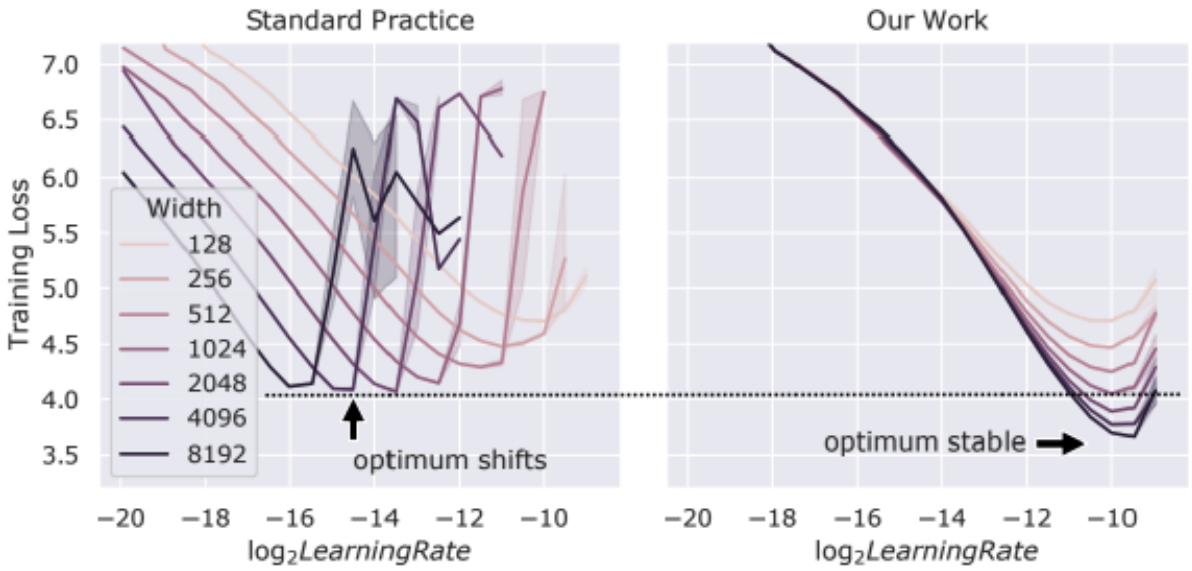


Figure 11: Comparison of LR tuning in standard practice vs. a scale-aware method. (Left) "Standard Practice" shows the optimal LR (the lowest point of the U-curve) shifts as model width increases. (Right) "Our Work" (using µP) shows the optimal LR remains stable for all model widths.

- **The Result:** When you do this, the "optimum is stable". All models, from Width 128 to 8192, achieve their best performance when the *base_lr* is set to the same value (approx. -10).
- **Conclusion:** This is the solution. It proves that if you implement the µP recipe, you can find the optimal **base_lr** (l) on a cheap, small model and be confident that this l (when used in the l/r formula) will be optimal for your giant model.

The µP Recipe (The Table) The table provides the specific rules for scaling hyperparameters from a base model (M) to a larger model (M') that is r times wider.

- **The Variables:**
  - M: Your small, base model (e.g., Width 128).
  - M': Your new, large model.
  - r: The scaling factor. (e.g., if M' has Width 1280, $r = 1280/128 = 10$).
  - l (Learning Rate), σ (Initialization Variance), τ (Multiplier): Hyperparameters you find on model M.
- **The Categories:** The recipe forces you to treat different parts of the model differently.
  - **"matrix-like":** The "engines" of the model. These are the large computational layers like FFNs and Attention projections, which grow with model width.
  - **"others":** Simpler parameters, like the embedding table, which do not scale in the same way.
- **The Key Rules:**

  - **Learning Rate Rule:** For "matrix-like" layers, the LR must be scaled by l/r. This is the core of the trick. A model that is 10x wider ($r = 10$) must use a 10x smaller *actual* LR for these layers.
  - **Initialization Rule:** The initialization variance for "matrix-like" layers must also be scaled by σ/r.
  - **Stability Rule:** The LR and Init. for "others" (like embeddings) *stay the same.*

- By following this specific recipe, µP "cancels out" the chaotic effects of scaling, making the optimal 'baselr' (l) stable.

3.6 Caution: scaling behaviors can differ downstream

This compares how scaling behaves for pre-training versus downstream evaluation. The two plots show that scaling can be predictable during pre-training but much less predictable on downstream tasks.

Left Plot: Pre-training scaling.

- **X-axis:** Number of model parameters (log-scale). Larger values mean larger models.
- **Y-axis:** Negative log-perplexity. Higher values mean lower perplexity, so better language modeling.
- **Points:** Each point corresponds to one trained model variant (e.g., NL6-XXL, NL12-XXL).
- **Trend:** As the number of parameters increases, negative log-perplexity increases smoothly. This means pre-training loss improves consistently with model size.
- **Interpretation:** Pre-training scaling is predictable. Larger models almost always achieve better pre-training loss. The relationship depends mainly on the parameter count.

Pre-training loss measures how well a language model predicts the next token during its initial large-scale training phase. This phase happens before the model is used for any downstream task.

Next-token prediction. During pre-training, the model receives a sequence of tokens

$$x_1, x_2, \ldots, x_{t-1},$$

and must predict the probability of the next token x_t.

Cross-entropy loss. The standard loss function is the negative log-likelihood:

$$\text{loss} = -\log p_\theta(x_t \mid x_{<t}),$$

and the total loss is the average of this value across all tokens in the dataset. A lower loss indicates that the model assigns higher probability to the correct next token.

Interpretation.

- Low loss means good next-token prediction.
- High loss means poor prediction.
- This loss is the main metric used to evaluate pre-training progress.

Negative log-perplexity. Perplexity is defined as

$$\text{Perplexity} = e^{\text{loss}}.$$

Taking the negative logarithm gives

$$-\log(\text{Perplexity}) = -\text{loss}.$$

Because lower loss is better, higher negative log-perplexity corresponds to better performance. This is why many pre-training scaling plots use negative log-perplexity on the y-axis: larger values indicate better language modeling.

Right Plot: Downstream scaling.

- **X-axis:** Number of model parameters (log-scale), same as the left plot.
- **Y-axis:** SuperGLUE accuracy. Higher values mean better downstream task performance.
- **Points:** Same family of models as in the left plot.
- **Trend:** Accuracy does not increase smoothly with model size. Some medium-sized models (e.g., NL32-XL) perform better than larger ones. Some very large models show little or no improvement.
- **Interpretation:** Downstream scaling is irregular. Model size alone does not reliably predict downstream accuracy. Architecture differences matter more, and performance varies across tasks.

Key takeaway.

- Pre-training scaling is smooth and strongly tied to model size.
- Downstream scaling is noisy and less predictable.
- We cannot assume "bigger is better" for downstream tasks.

3.7 Some surprising takeaways

This highlights a key observation from scaling law research: the effect of hyperparameters on large language models can often be predicted before training the large model. The reason is that many hyperparameter behaviors remain stable across scales.

Predictable hyperparameter effects.

- **Optimizer choice** (e.g., Adam vs. SGD).
- **Model depth** (how increasing the number of layers changes performance).
- **Architecture choice** (which architecture continues to improve as it scales).

These factors show consistent scaling trends when measured on small models. This allows us to identify the best settings without running many large-scale experiments.

Design procedure based on scaling laws.

1. Train several small models using different hyperparameters.
2. Fit a scaling law (for example, comparing the scaling curves of Adam and SGD).
3. Select the optimal hyperparameters for the large model using the scaling prediction.

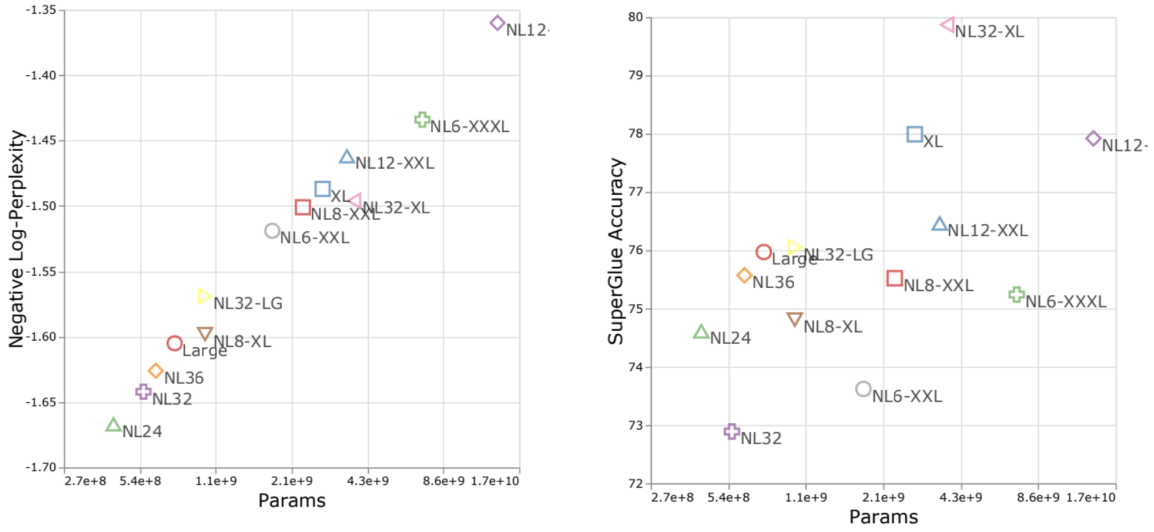


Figure 12: Pre-training shows smooth, predictable scaling, but downstream accuracy varies significantly with model size.

4 Joint laws and compute trade-offs

4.1 One important use of scaling laws

This addresses an important question: **do we need more data or a bigger model?** The loss curves show that small models cannot fully use large datasets. When the model is too small, additional data produces little improvement.

Loss vs. Model and Dataset Size (top-right).

- **Axes:** X-axis is dataset size (tokens, log-scale). Y-axis is loss.
- **Colors:** Each curve corresponds to a different model size.
- **Trend:** Bigger models achieve lower loss at the same data size. For any model, more data decreases loss.
- **Interpretation:** Loss depends on both model size and dataset size. Small models saturate early and waste large datasets.

Joint data–model scaling laws. These describe how dataset size and model size together determine error.

Rosenfeld et al. (2020)

$$\text{Error} = n^{-\alpha} + m^{-\beta} + C$$

Meaning of each variable.

- n: Dataset size (number of training examples or tokens).
- m: Model size (number of parameters).
- α: Data scaling exponent. Controls how fast error decreases as n increases.

- β: Model scaling exponent. Controls how fast error decreases as m increases.
- C: Irreducible error floor.

Interpretation.

- $n^{-\alpha}$ shows how much improvement comes from more data.
- $m^{-\beta}$ shows how much improvement comes from a larger model.
- The two effects add together. If either n or m is too small, error remains high.

Kaplan et al. (2020)

$$\text{Error} = \left[m^{-\alpha} + n^{-1}\right]^{\beta}$$

Meaning of each variable.

- $m^{-\alpha}$: Benefit from increasing model size.
- n^{-1}: Benefit from increasing dataset size (fixed exponent of 1).
- β: Joint exponent that controls the overall shape of the curve.

Interpretation.

- Error depends on a *combined* term: model quality + data quality.
- If either m or n is small, the sum inside the brackets is large, so total error is large.
- This form captures interaction between model size and data size.

3D Loss Landscape (bottom-right).

- **Axes:** X-axis: log data fraction, Y-axis: log model fraction, Z-axis: log error.
- **Trend:** Increasing either data or model size reduces error.
- **Interpretation:** The shape matches the predictions of the joint scaling laws.

Key takeaway. Joint scaling laws explain how to balance model size and dataset size. They show why data is wasted on small models, and why scaling must be balanced.

4.2 Model–data joint scaling is accurate

This shows that the joint data–model scaling law

$$\text{Error}(n, m) = n^{-\alpha} + m^{-\beta} + C$$

can accurately predict performance for many combinations of dataset size and model size.

Meaning of each variable.

- n: dataset size (number of training examples or tokens).
- m: model size (number of parameters).
- α: data scaling exponent, controlling how fast error decreases when n increases.

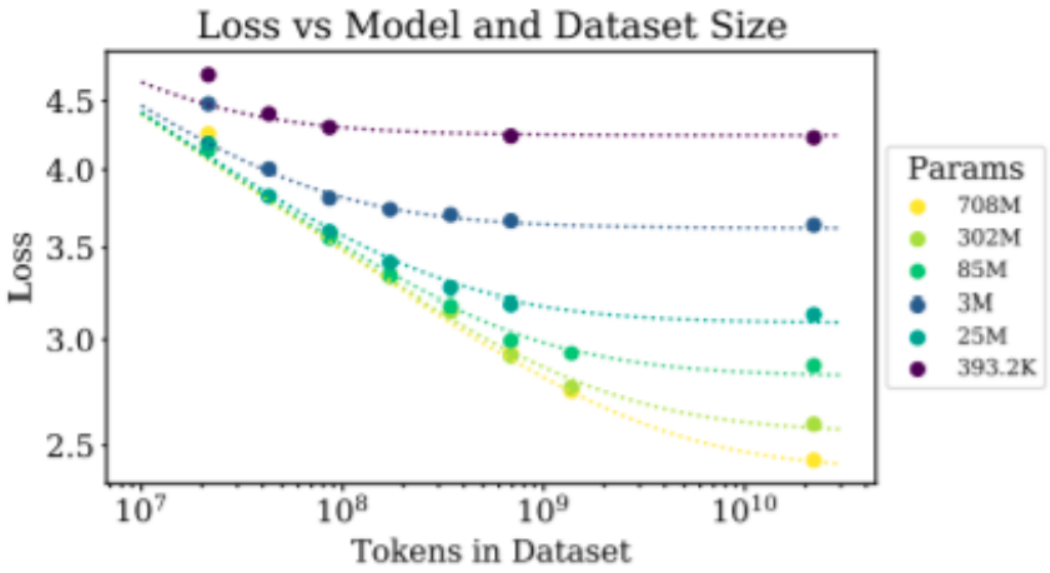


Figure 13: Joint data–model scaling laws showing how dataset size and model size interact.

- β: model scaling exponent, controlling how fast error decreases when m increases.
- C: irreducible error floor.

The idea in Rosenfeld et al. is: fit α, β, C using *small* models and *small* datasets, then use the formula above to predict performance everywhere else.

(a) Illustration (left panel).

- **Axes:** X-axis is data fraction $\log_2(n/N)$; Y-axis is model fraction $\log_2(m/M)$, where N and M are the largest dataset and largest model in the study.
- **Green circles:** Points where the model is actually trained and error is measured. These are small n and small m settings.
- **Red circles:** Points that will be *predicted* using the fitted scaling law, without training at those settings.

(b) Extrapolation on ImageNet (middle panel).

- **Axes:** X-axis is the *measured* top-1 error of a model (from real training). Y-axis is the *estimated* top-1 error predicted by the scaling law.
- **Diagonal line:** Perfect prediction line (estimated error = measured error).
- **Green dots:** Points used for fitting (small n, small m).
- **Red dots:** Extrapolated predictions for larger models and datasets.
- **Trend:** Both green and red dots lie close to the diagonal. The small bias μ and variance σ printed in the legend show that prediction error is only a few percent.

- **Interpretation:** Once the exponents α and β are fitted on small-scale runs, the same law predicts ImageNet performance very accurately at much larger scales.

(c) Extrapolation on WikiText-103 (right panel).

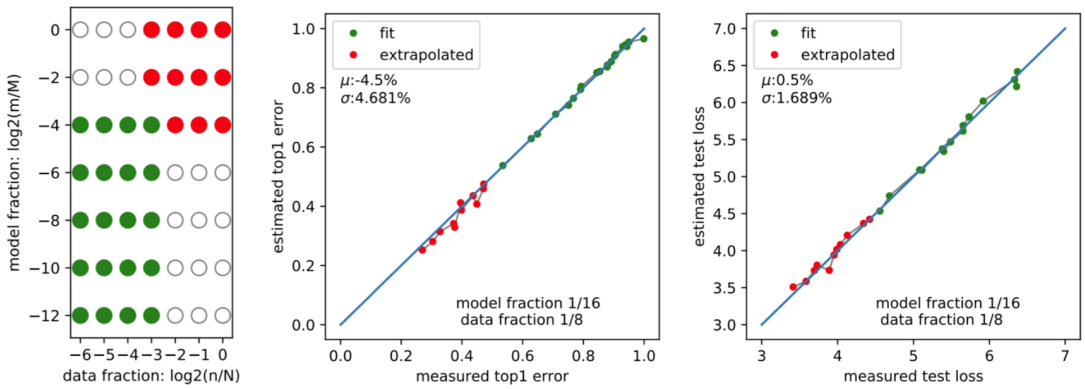
- **Axes:** X-axis is the measured test loss; Y-axis is the estimated test loss from the scaling law.
- **Diagonal line:** Perfect agreement between measurement and prediction.
- **Dots:** Green (fit) and red (extrapolated) points are both close to the diagonal, with very small μ and σ.
- **Interpretation:** The same joint law generalizes well to a language modeling task, not just vision.

Key takeaway.

- We can fit $n^{-\alpha} + m^{-\beta} + C$ using only a small subset of model sizes and data sizes.
- The resulting law accurately predicts the error at much larger scales.
- This allows us to trade off data and model size by optimizing

$$n^{-\alpha} + m^{-\beta} + C$$

  subject to our compute and cost constraints.



(a) Illustration. (b) Extrapolation on ImageNet (c) Extrapolation on WikiText-103.

Figure 14: Joint data–model scaling fitted on small models and datasets produces accurate predictions for larger settings on both ImageNet and WikiText-103.

4.3 Compute tradeoffs

This examines how model performance depends on compute. Even if we know how performance scales with model size and dataset size, we must still decide how to allocate a fixed compute budget. The key question is:

Is it better to use a big model that is undertrained, or a smaller model that is fully trained?

The two plots illustrate this tradeoff.

(Left) Performance vs. Compute Budget (Kaplan et al. 2021).

- **Axes:** X-axis: number of *non-embedding* parameters Y-axis: test loss
- **Colors:** Each curve corresponds to a fixed compute budget measured in PF-days (PetaFLOP $\times$ days). Brighter colors indicate more compute.
- **Trend:** For each fixed budget, test loss decreases as model size increases, reaches an optimal point, then increases again. The right side rises sharply because large models are severely undertrained.
- **Interpretation:** When compute is limited, the largest model is not the best choice. There is an *optimal model size* for each compute budget.

(Right) Validation Loss vs. Compute (Brown et al. 2020).

- **Axes:** X-axis: compute (PetaFLOP/s-days, log-scale) Y-axis: validation loss
- **Colors:** Each color represents a different model size.
- **Trend:** Each model follows a smooth curve: as compute increases, validation loss decreases. The optimal model size increases with compute. The dashed line is a simple power-law fit: $L = 2.57 \cdot C^{-0.048}$.
- **Interpretation:** For small compute budgets, small models perform better. For large compute budgets, larger models eventually dominate.

Key takeaway.

- Compute places a hard constraint on scaling.
- Larger models require more compute to reach their potential.
- Scaling laws help determine the best model size for a given budget.
- A compute-constrained large model will underperform a fully trained smaller model.

4.4 Caution: "Optimal" scaling laws are hard to get

This explains that estimating the *optimal* relationship between model size, compute, and performance is difficult. Earlier works (Rosenfeld and Kaplan) propose formulas that connect these quantities, but later analysis from Hoffman et al. ("Chinchilla") shows that the fits can be noticeably inaccurate.

The plot compares the predicted optimal scaling curves with real large models.

Plot interpretation.

- **Axes:** X-axis: total compute (FLOPs, log-scale). Y-axis: number of parameters (log-scale).
- **Dashed black line:** The optimal scaling rule proposed by Kaplan et al. (2020). This line specifies how many parameters a model should have for a given compute budget.

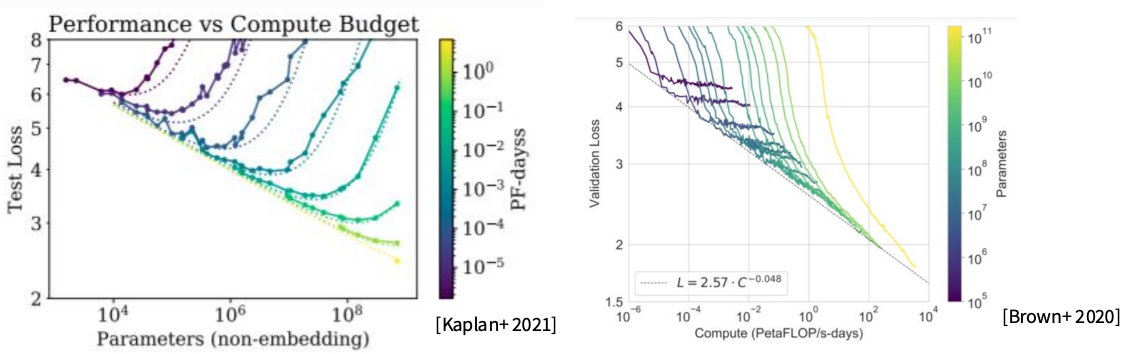


Figure 15: Compute–performance tradeoffs: for each compute budget, scaling laws identify the optimal model size.

- **Colored solid curves (Approach 1, 2, 3):** Alternative fits proposed by Hoffman et al. (Chinchilla). These are revised estimates of the optimal compute–parameter tradeoff.
- **Blue points:** Real models from Kaplan et al.'s scaling dataset.
- **Stars:** Major large-scale models: Chinchilla (70B), Gopher (280B), GPT-3 (175B), and Megatron-Turing (530B). These points represent real trained models and where they fall relative to the predicted curves.

Observed trends.

- The Kaplan dashed line overestimates the optimal parameter count for a given compute budget. It predicts much larger models than what later experiments show to be efficient.
- The Chinchilla fits (solid curves) lie below the Kaplan line. This means the optimal model is *smaller* than previously believed.

Interpretation.

- Kaplan's original scaling law for optimal model size is inaccurate.
- Chinchilla argues that the compute–parameter relationship must be revised, with more emphasis on training tokens rather than only model size.
- This also explains why Chinchilla (70B) outperforms much larger models like GPT-3 (175B) with the same compute: it is closer to the true optimal scaling point.

Key takeaway. Estimating the "optimal" model size for a given compute budget is hard. Earlier scaling laws can be significantly off, and updated analyses show that models should often be *smaller* and trained on more data.

4.5 Main difference: accounting for learning rate schedules

This shows that differences in learning rate (LR) schedules can create apparent differences in scaling behavior. To compare models fairly, we must account for how the LR decays over training.

The figure contains two rows of three plots. Each row uses a different total training length, and each color corresponds to a different cosine LR decay schedule.

(Left column) Learning rate schedules.

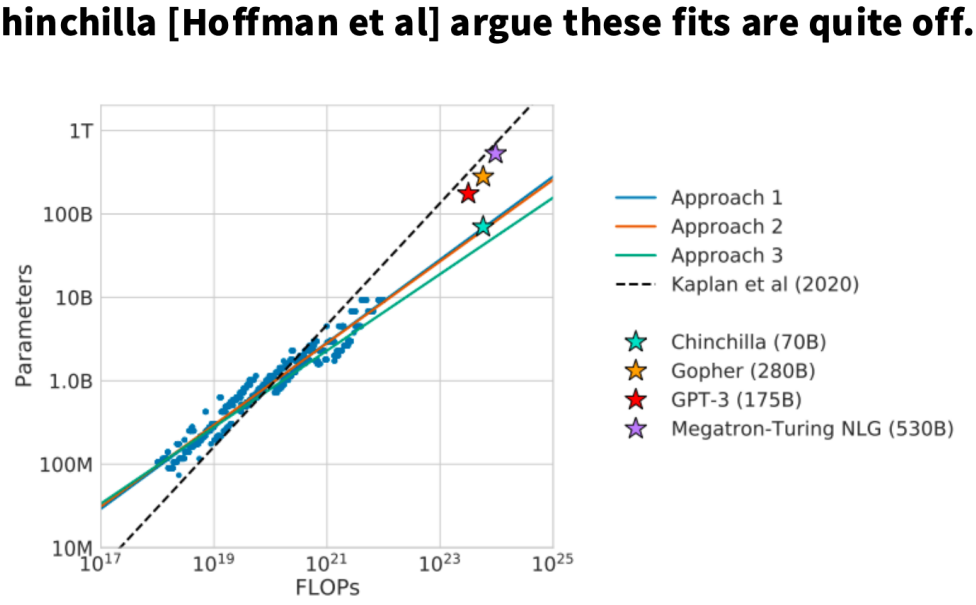


Figure 16: Comparison of scaling-law fits. Chinchilla shows that earlier predictions for optimal model size are inaccurate.

- **Axes:** X-axis: training progress in "million sequences" Y-axis: learning rate normalized by the maximum LR
- **Curves:** Each curve is a cosine decay with a different cycle length: 1.0×, 1.1×, 1.25×, 1.5×, 2.0×, and 5.0× number of steps.
- **Trend:** Shorter cycles (e.g., 1.0×) decay quickly. Longer cycles (e.g., 5.0×) maintain high LR for longer.
- **Interpretation:** Changing only the LR schedule—even with the same model and dataset—changes the training dynamics.

(Middle column) Training loss curves.

- **Axes:** X-axis: million sequences Y-axis: training loss
- **Trend:** Curves diverge at the beginning due to different LR decay speeds. Slower LR decay (long cycles) leads to higher loss early on. As training continues, curves gradually converge.
- **Interpretation:** Comparing training curves without matching LR schedules can make two similar models appear very different.

(Right column) C4 loss curves.

- **Axes:** X-axis: million sequences Y-axis: C4 validation loss
- **Trend:** The separation between curves is clearer here. Models with longer LR cycles maintain higher loss throughout training. Shorter cycles reach lower loss earlier.

- **Interpretation:** LR schedule has a strong effect on downstream evaluation loss. Without controlling for LR decay, scaling comparisons can be misleading.

Key takeaway.

- Learning rate schedules significantly influence both training and validation loss curves.
- Apparent differences in scaling may come only from different LR schedules.
- Fair scaling comparisons require matching not just model and data size, but also LR schedule.

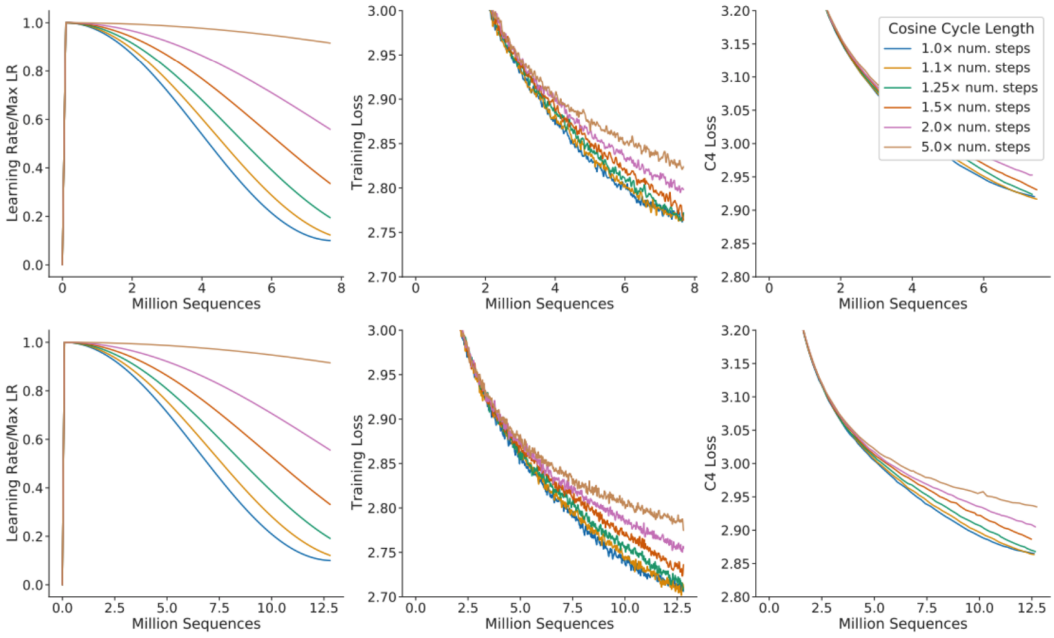


Figure 17: Learning rate schedules cause differences in loss curves. Understanding these effects is necessary for fair scaling comparisons.

4.6 Chinchilla in depth: three methods

This summarizes the three methods used by the Chinchilla authors (Hoffmann et al.) to estimate optimal scaling laws. Each method fits the relationship between model size, dataset size, and compute in a different way.

The table reports two coefficients:

$$N_{\text{opt}} \propto C^a, \quad D_{\text{opt}} \propto C^b$$

where:

- N_{opt}: optimal model size (number of parameters),
- D_{opt}: optimal dataset size (number of tokens),

- C: total compute budget,
- a: how model size should grow with compute,
- b: how dataset size should grow with compute.

High-level intuition. We want to know: if compute increases, how should model size and data size grow to stay compute-optimal? These exponents a and b estimate those growth rates.

Three methods.

- **1. Minimum over training curves:** Look at all training runs and extract the minimum loss seen at each FLOP level.
- **2. IsoFLOP profiles:** For a fixed FLOP budget, vary model size and training tokens, then find the minimum-loss configuration.
- **3. Parametric modelling of the loss:** Fit a full parametric loss function to all data (more assumptions).

Observation. Methods 1 and 2 give very similar exponents ($a \approx 0.5$), while method 3 is less stable. Kaplan et al. predicted $a = 0.73$, which Chinchilla argues is too large.

4.7 Method 1: minimum over runs

This method uses the envelope of the best-performing points from all training curves.

High-level intuition. For any compute value (FLOPs), some model–data pair achieves the lowest loss. Plotting these "best points" forms a clean power-law curve.

Left plot: training curve envelope.

- **Axes:** FLOPs vs. training loss.
- **Colors:** different model sizes (70M to 10B).
- **Trend:** The lower boundary of these curves forms a smooth downward power law.
- **Intuition:** Even though each training run has noise and LR differences, the best-loss envelope reveals the true scaling law.

Center plot: optimal model size vs. compute.

- Extract the parameter count that lies on the minimum-loss envelope for each FLOP value.
- These points align closely with a power law $N_{\text{opt}} \propto C^a$.

Right plot: optimal token count vs. compute.

- Same process, but using dataset size.
- Produces a similar power-law relationship $D_{\text{opt}} \propto C^b$.

Conclusion. The envelope method gives clean, stable scaling exponents with minimal assumptions.

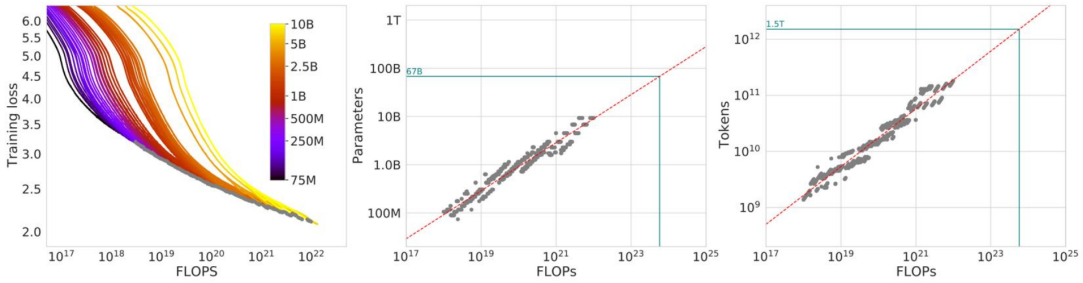


Figure 18: Method 1 extracts a power law by taking the minimum loss over all training curves.

4.8 Method 2: IsoFLOPs

IsoFLOP analysis fixes a compute budget and varies model size and dataset size to find the minimum training loss achievable at that budget.

High-level intuition. For a fixed compute budget, making the model too large undertrains it. Making it too small wastes compute. There is a "valley" where loss is minimized — the optimal tradeoff.

Left plot: IsoFLOP curves.

- **Axes:** parameters vs. training loss.
- **Different curves:** each curve corresponds to one FLOP budget.
- **Trend:** Each curve is convex (U-shaped). Loss is high for small models (underfitting) and high for large models (undertraining).
- **Minimum:** The bottom of each U-shape is the optimal model size.

Center and right plots. Plotting the minima from each IsoFLOP curve gives clean power laws:

$$N_{\text{opt}} \propto C^a, \qquad D_{\text{opt}} \propto C^b.$$

Conclusion. IsoFLOP valleys provide another way to identify optimal model–data balance, confirming method 1.

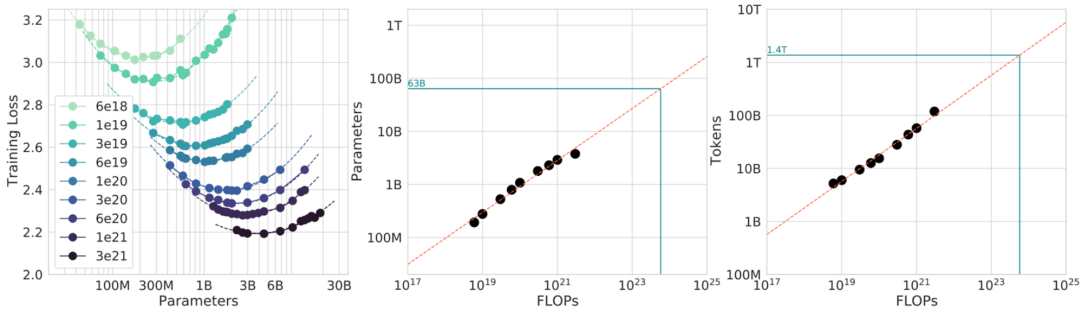


Figure 19: Method 2 uses IsoFLOP valleys to estimate optimal model and data sizes for each compute budget.

4.9 Method 3: joint fits

This method fits a full parametric model to the loss surface as a function of model size N and dataset size D.

High-level intuition. Instead of taking minima, this method fits the entire loss landscape. This can be very powerful, but also sensitive to noise and modelling choices.

Left plot: IsoLoss contours.

- **Axes:** FLOPs (x-axis) and model size (y-axis).
- **Contours:** lines of constant loss.
- **Efficient frontier:** The blue line shows, for each FLOP value, the smallest model size that achieves that loss.
- **Intuition:** This traces the optimal model size across compute.

Right plot: IsoFLOP slices.

- **Axes:** model size vs. loss for fixed FLOP budgets.
- **Trend:** U-shaped curves similar to Method 2.
- The parametric fit tries to match these shapes across all budgets.

Conclusion. Joint fitting is elegant but sensitive. Later work shows this method was unstable in the original Chinchilla paper.

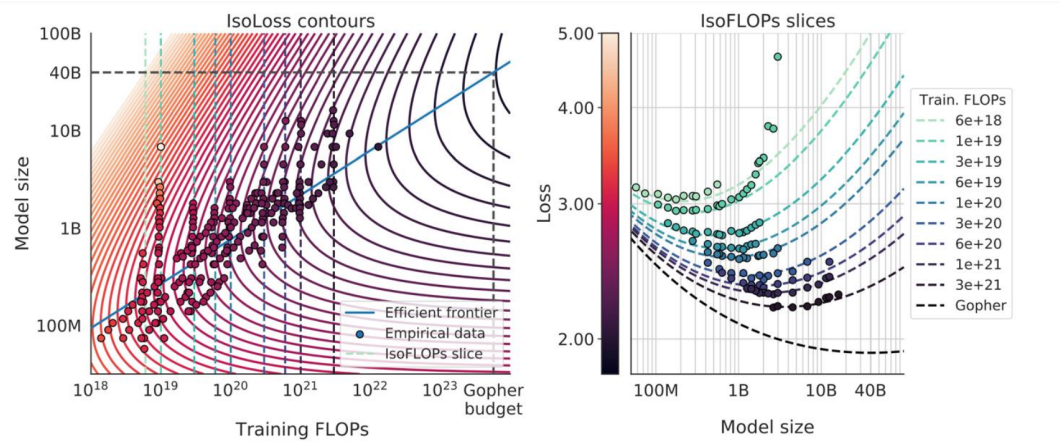


Figure 20: Method 3 fits the full loss surface using all model–data points.

4.10 Addendum: errors in Chinchilla Method 3

Later analysis (Besirolu et al. 2024) showed that Method 3 had issues. Some raw data was misprocessed, and the parametric fit became inconsistent with Methods 1 and 2.

Left plot: residuals.

- **Axes:** residual error (difference between predicted and measured loss).

- **Green distribution:** residuals reported in Hoffmann et al.
- **Blue distribution:** corrected residuals after data recovery.
- **Intuition:** A good fit should have residuals centered around zero. The corrected fit is more centered and tighter.

Right plot: optimal tokens per parameter.

- **Axes:** compute budget vs. optimal token/parameter ratio.
- **Green line:** Hoffmann et al. (Method 3).
- **Blue line:** corrected analysis.
- **Observation:** The corrected curve matches Method 1 and Method 2, fixing inconsistencies.

High-level intuition. The original Method 3 likely had data-quality issues. After correction, all three methods agree:

$$N_{\text{opt}} \propto C^{0.5}, \qquad D_{\text{opt}} \propto C^{0.5}.$$

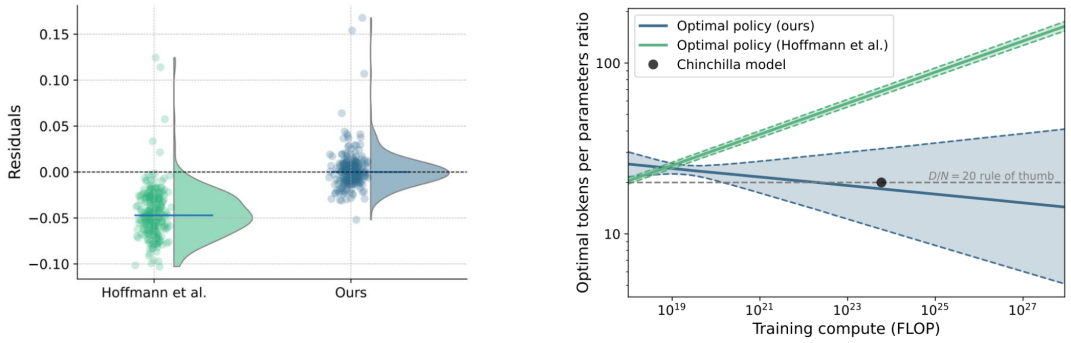


Figure 21: Corrected Method 3 results match Methods 1 and 2, fixing inconsistencies in the original Chinchilla paper.