# Scaling Laws for Neural Language Models

# Scaling Laws for Neural Language Models

**Jared Kaplan** [*]
Johns Hopkins University, OpenAI
jaredk@jhu.edu

**Sam McCandlish**[*]
OpenAI
sam@openai.com

**Tom Henighan**
OpenAI
henighan@openai.com

**Tom B. Brown**
OpenAI
tom@openai.com

**Benjamin Chess**
OpenAI
bchess@openai.com

**Rewon Child**
OpenAI
rewon@openai.com

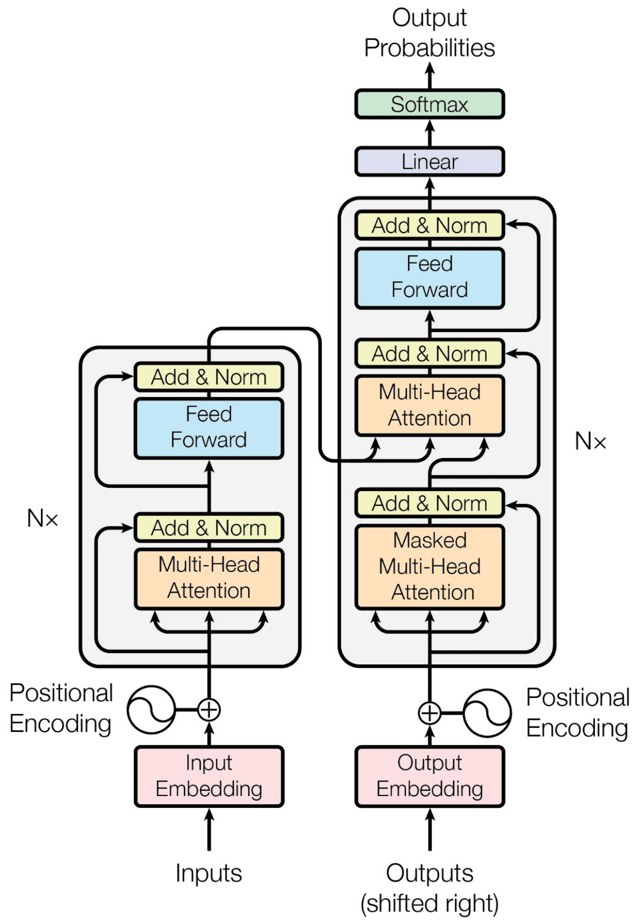
**Scott Gray**
OpenAI
scott@openai.com

**Alec Radford**
OpenAI
alec@openai.com

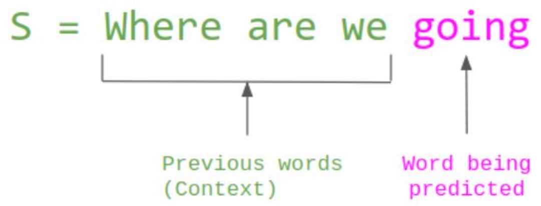
**Jeffrey Wu**
OpenAI
jeffwu@openai.com

**Dario Amodei**
OpenAI
damodei@openai.com

## Transformer architecture



## Loss function for language models



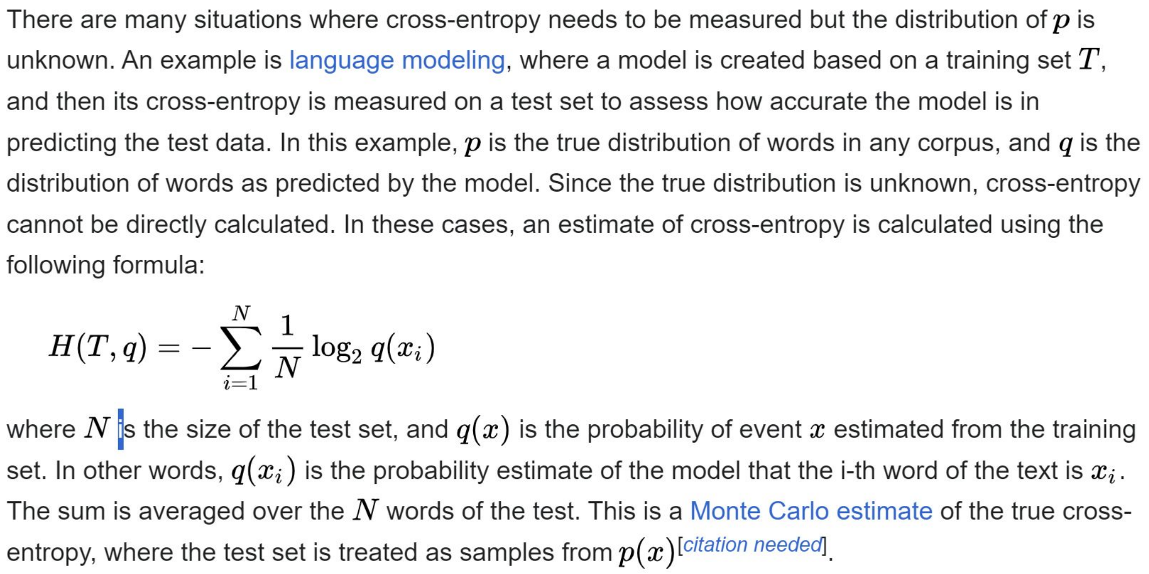
P(S) = P(Where) x P(are | Where) x P(we | Where are) x P(going | Where are we)

Probability of a sentence can be defined as the product of the probability of each symbol given the previous symbols

cross-entropy loss is measuring if the model gives the true “next word” a high probability. The lower the loss, the higher the model probability of the true “next word”. here

# Cross Entropy Loss function

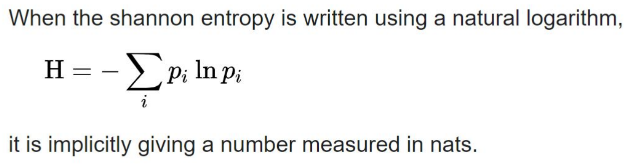
- **Cross-entropy** is a popular loss function used in machine learning to measure the performance of a classification model. Namely, it measures the difference between the discovered probability distribution of a classification model and the predicted values.



There are many situations where cross-entropy needs to be measured but the distribution of $p$ is unknown. An example is language modeling, where a model is created based on a training set $T$, and then its cross-entropy is measured on a test set to assess how accurate the model is in predicting the test data. In this example, $p$ is the true distribution of words in any corpus, and $q$ is the distribution of words as predicted by the model. Since the true distribution is unknown, cross-entropy cannot be directly calculated. In these cases, an estimate of cross-entropy is calculated using the following formula:

$$H(T,q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

where $N$ is the size of the test set, and $q(x)$ is the probability of event $x$ estimated from the training set. In other words, $q(x_i)$ is the probability estimate of the model that the i-th word of the text is $x_i$. The sum is averaged over the $N$ words of the test. This is a Monte Carlo estimate of the true cross-entropy, where the test set is treated as samples from $p(x)$[citation needed].



When the shannon entropy is written using a natural logarithm,

$$\mathrm{H} = -\sum_i p_i \ln p_i$$

it is implicitly giving a number measured in nats.

## Cross-entropy loss

- **Entropy** calculates the degree of randomness or disorder within a system to measure the uncertainty of an event. If an outcome is certain, the measure of entropy will be low.
- **Cross-entropy** is a popular loss function used in machine learning to measure the performance of a classification model. Namely, it measures the difference between the discovered probability distribution of a classification model and the predicted values. When applied to binary classification tasks, it is commonly referred to as log loss.
- **Binary cross-entropy** is used when performing binary classification, and categorical cross-entropy is used for multi-class classification.
- **Cross-entropy is similar to KL divergence**, but they serve different purposes: cross-entropy is typically used in machine learning to evaluate the performance of a model where the objective is to minimize the error between the predicted probability distribution and true distribution, whereas KL is more useful in unsupervised learning tasks where the objective is to uncover structure in data by minimizing the divergence between the true and learned data distributions.

Given constrained compute budget measured in FLOPs, floating-point operations, what would be the optimal combination of model size and training data size (measured in number of tokens) that yields the lowest loss?

$C = 6NBS$

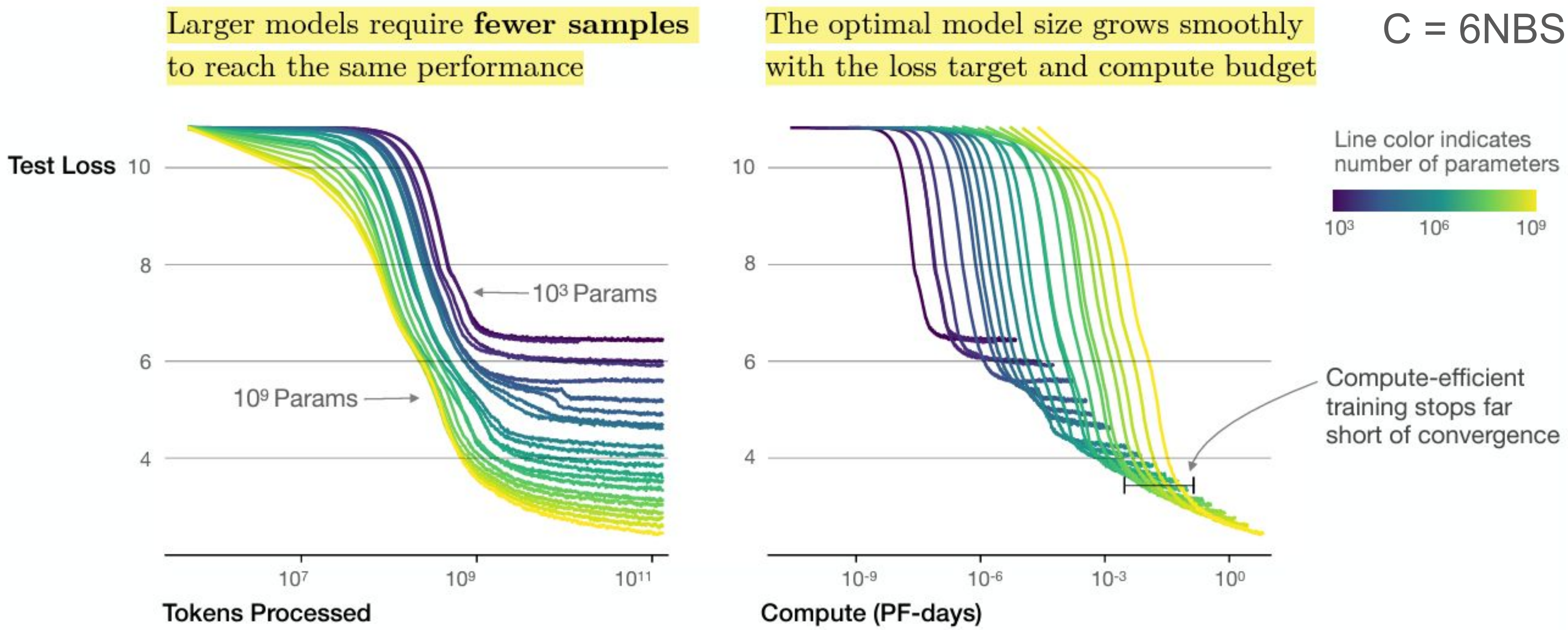


**Figure 2** We show a series of language model training runs, with models ranging in size from $10^3$ to $10^9$ parameters (excluding embeddings).

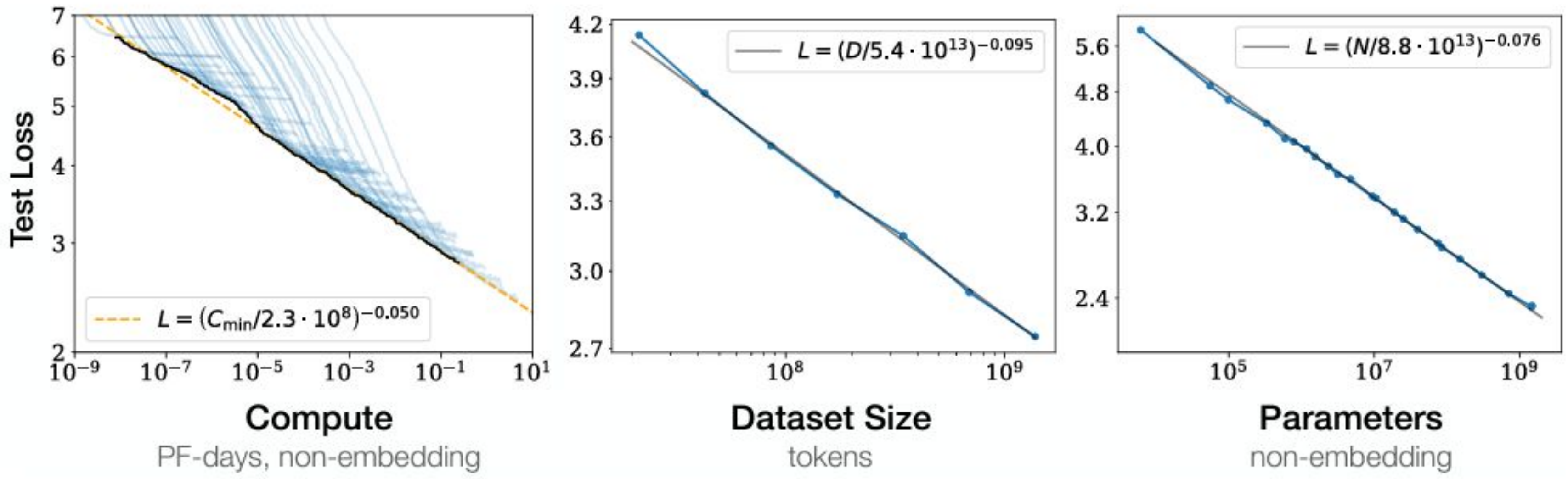
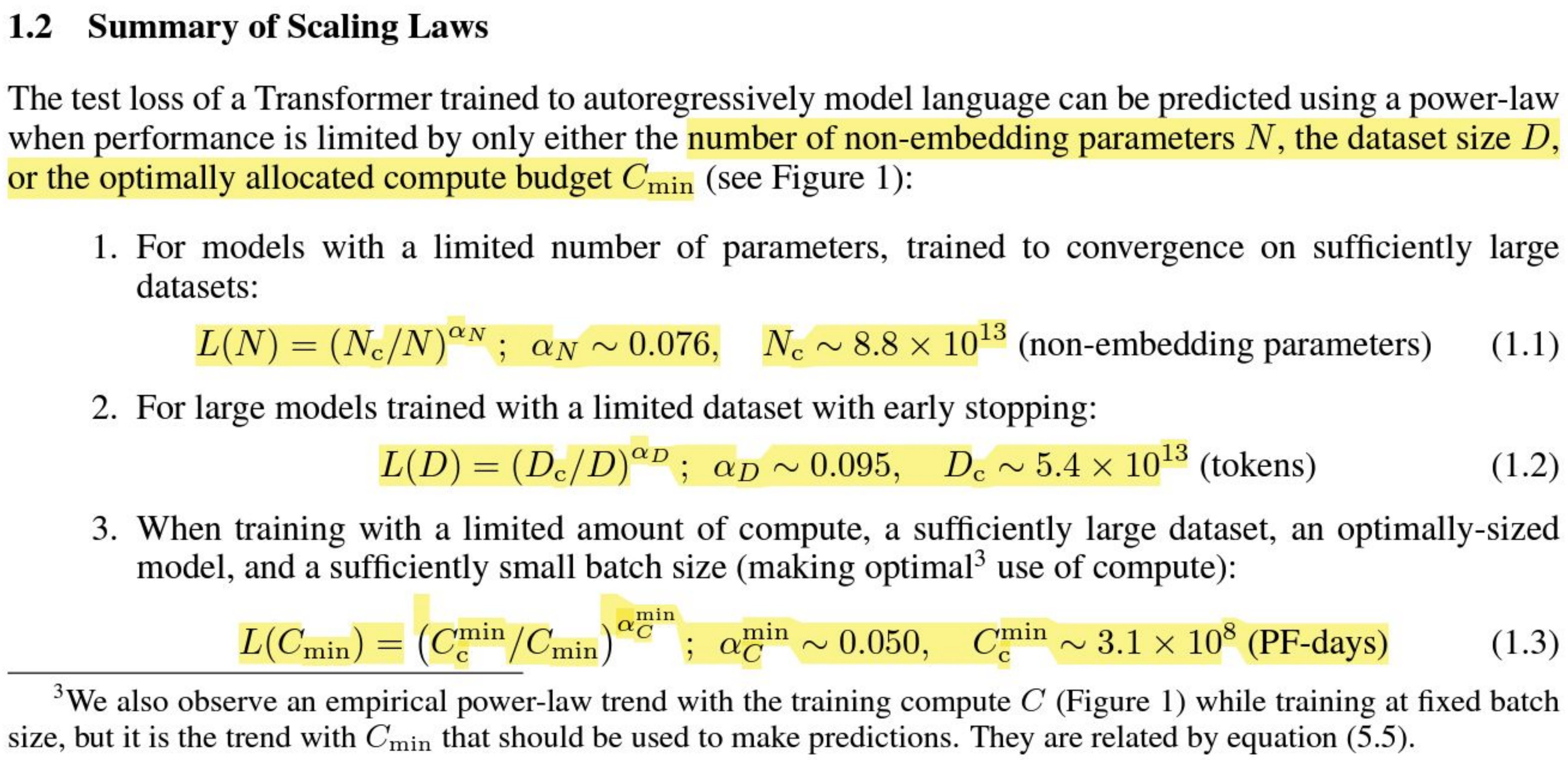


**Figure 1** Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.



## 1.2 Summary of Scaling Laws

The test loss of a Transformer trained to autoregressively model language can be predicted using a power-law when performance is limited by only either the number of non-embedding parameters $N$, the dataset size $D$, or the optimally allocated compute budget $C_{\rm min}$ (see Figure 1):

1. For models with a limited number of parameters, trained to convergence on sufficiently large datasets:
$$L(N) = (N_{\rm c}/N)^{\alpha_N}\ ;\ \ \alpha_N \sim 0.076, \quad N_{\rm c} \sim 8.8 \times 10^{13} \text{ (non-embedding parameters)} \tag{1.1}$$
2. For large models trained with a limited dataset with early stopping:
$$L(D) = (D_{\rm c}/D)^{\alpha_D}\ ;\ \ \alpha_D \sim 0.095, \quad D_{\rm c} \sim 5.4 \times 10^{13} \text{ (tokens)} \tag{1.2}$$
3. When training with a limited amount of compute, a sufficiently large dataset, an optimally-sized model, and a sufficiently small batch size (making optimal[3] use of compute):
$$L(C_{\rm min}) = \left(C_{\rm c}^{\rm min}/C_{\rm min}\right)^{\alpha_C^{\rm min}}\ ;\ \ \alpha_C^{\rm min} \sim 0.050, \quad C_{\rm c}^{\rm min} \sim 3.1 \times 10^{8} \text{ (PF-days)} \tag{1.3}$$

[3]We also observe an empirical power-law trend with the training compute $C$ (Figure 1) while training at fixed batch size, but it is the trend with $C_{\rm min}$ that should be used to make predictions. They are related by equation (5.5).

Our choice of L(N,D) satisfies the first requirement because we can rescale Nc,Dc with changes in the vocabulary. This also implies that the values of Nc,Dc have no fundamental meaning.

## Scaling law

$$L(N, D) = \left[\left(\frac{N_c}{N}\right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D}\right]^{\alpha_D}$$

| Parameter | $\alpha_N$ | $\alpha_D$ | $N_c$ | $D_c$ |
|---|---|---|---|---|
| Value | 0.076 | 0.103 | $6.4 \times 10^{13}$ | $1.8 \times 10^{13}$ |

**Table 2** Fits to $L(N, D)$

To avoid overfitting

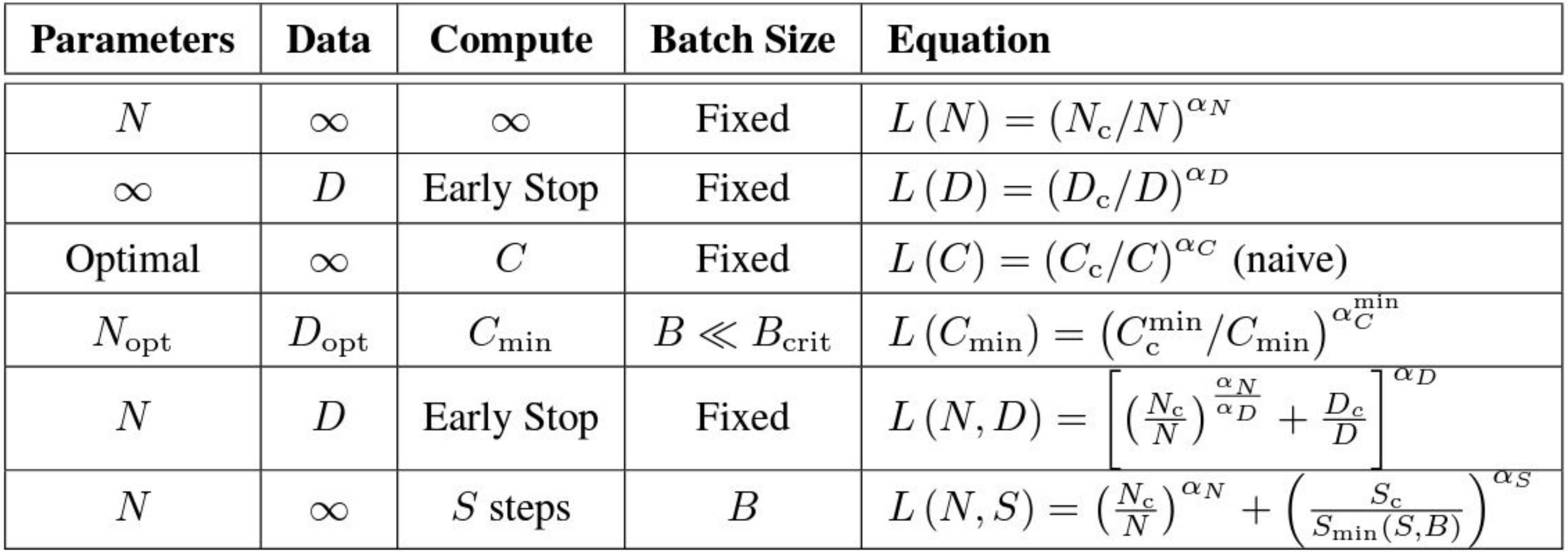
$$D \gtrsim (5 \times 10^3)\, N^{0.74}$$

**Performance depends strongly on scale, weakly on model shape:** Model performance depends most strongly on scale, which consists of three factors: the number of model parameters $N$ (excluding embeddings), the size of the dataset $D$, and the amount of compute $C$ used for training. Within reasonable limits, performance depends very weakly on other architectural hyperparameters such as depth vs. width. (Section 3)

We use the following notation:

- $L$ – the cross entropy loss in nats. Typically it will be averaged over the tokens in a context, but in some cases we report the loss for specific tokens within the context.
- $N$ – the number of model parameters, *excluding all vocabulary and positional embeddings*
- $C \approx 6NBS$ – an estimate of the total non-embedding training compute, where $B$ is the batch size, and $S$ is the number of training steps (ie parameter updates). We quote numerical values in PF-days, where one PF-day $= 10^{15} \times 24 \times 3600 = 8.64 \times 10^{19}$ floating point operations.
- $D$ – the dataset size in tokens
- $B_{\text{crit}}$ – the critical batch size [MKAT18], defined and discussed in Section 5.1. Training at the critical batch size provides a roughly optimal compromise between time and compute efficiency.
- $C_{\text{min}}$ – an estimate of the minimum amount of non-embedding compute to reach a given value of the loss. This is the training compute that would be used if the model were trained at a batch size much less than the critical batch size.
- $S_{\text{min}}$ – an estimate of the minimal number of training steps needed to reach a given value of the loss. This is also the number of training steps that would be used if the model were trained at a batch size much greater than the critical batch size.
- $\alpha_X$ – power-law exponents for the scaling of the loss as $L(X) \propto 1/X^{\alpha_X}$ where $X$ can be any of $N, D, C, S, B, C^{\text{min}}$.

## Summary of Power Laws



| **Parameters** | **Data** | **Compute** | **Batch Size** | **Equation** |
|---|---|---|---|---|
| $N$ | $\infty$ | $\infty$ | Fixed | $L(N) = (N_{\mathrm{c}}/N)^{\alpha_N}$ |
| $\infty$ | $D$ | Early Stop | Fixed | $L(D) = (D_{\mathrm{c}}/D)^{\alpha_D}$ |
| Optimal | $\infty$ | $C$ | Fixed | $L(C) = (C_{\mathrm{c}}/C)^{\alpha_C}$ (naive) |
| $N_{\mathrm{opt}}$ | $D_{\mathrm{opt}}$ | $C_{\mathrm{min}}$ | $B \ll B_{\mathrm{crit}}$ | $L(C_{\mathrm{min}}) = \left(C_{\mathrm{c}}^{\mathrm{min}}/C_{\mathrm{min}}\right)^{\alpha_C^{\mathrm{min}}}$ |
| $N$ | $D$ | Early Stop | Fixed | $L(N, D) = \left[\left(\frac{N_{\mathrm{c}}}{N}\right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D}\right]^{\alpha_D}$ |
| $N$ | $\infty$ | $S$ steps | $B$ | $L(N, S) = \left(\frac{N_{\mathrm{c}}}{N}\right)^{\alpha_N} + \left(\frac{S_{\mathrm{c}}}{S_{\mathrm{min}}(S,B)}\right)^{\alpha_S}$ |

**Table 4**

| **Power Law** | **Scale (tokenization-dependent)** |
|---|---|
| $\alpha_N = 0.076$ | $N_{\mathrm{c}} = 8.8 \times 10^{13}$ params (non-embed) |
| $\alpha_D = 0.095$ | $D_{\mathrm{c}} = 5.4 \times 10^{13}$ tokens |
| $\alpha_C = 0.057$ | $C_{\mathrm{c}} = 1.6 \times 10^{7}$ PF-days |
| $\alpha_C^{\mathrm{min}} = 0.050$ | $C_{\mathrm{c}}^{\mathrm{min}} = 3.1 \times 10^{8}$ PF-days |
| $\alpha_B = 0.21$ | $B_{*} = 2.1 \times 10^{8}$ tokens |
| $\alpha_S = 0.76$ | $S_{\mathrm{c}} = 2.1 \times 10^{3}$ steps |

**Table 5**

| **Compute-Efficient Value** | **Power Law** | **Scale** |
|---|---|---|
| $N_{\text{opt}} = N_e \cdot C_{\text{min}}^{p_N}$ | $p_N = 0.73$ | $N_e = 1.3 \cdot 10^9$ params |
| $B \ll B_{\text{crit}} = \frac{B_*}{L^{1/\alpha_B}} = B_e C_{\text{min}}^{p_B}$ | $p_B = 0.24$ | $B_e = 2.0 \cdot 10^6$ tokens |
| $S_{\text{min}} = S_e \cdot C_{\text{min}}^{p_S}$ (lower bound) | $p_S = 0.03$ | $S_e = 5.4 \cdot 10^3$ steps |
| $D_{\text{opt}} = D_e \cdot C_{\text{min}}^{p_D}$ (1 epoch) | $p_D = 0.27$ | $D_e = 2 \cdot 10^{10}$ tokens |

**Table 6**

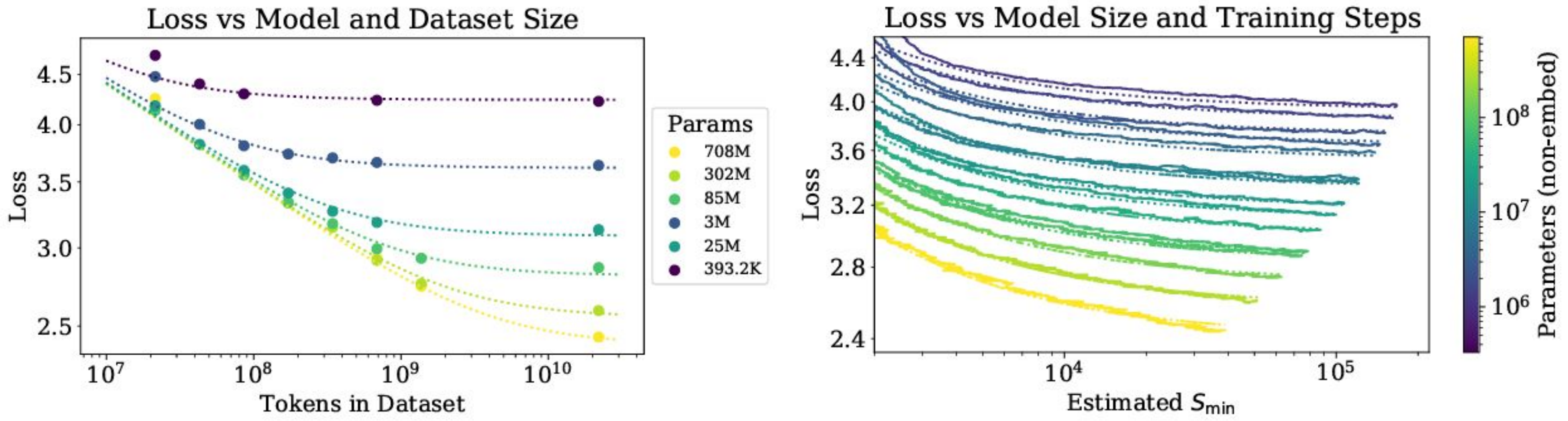
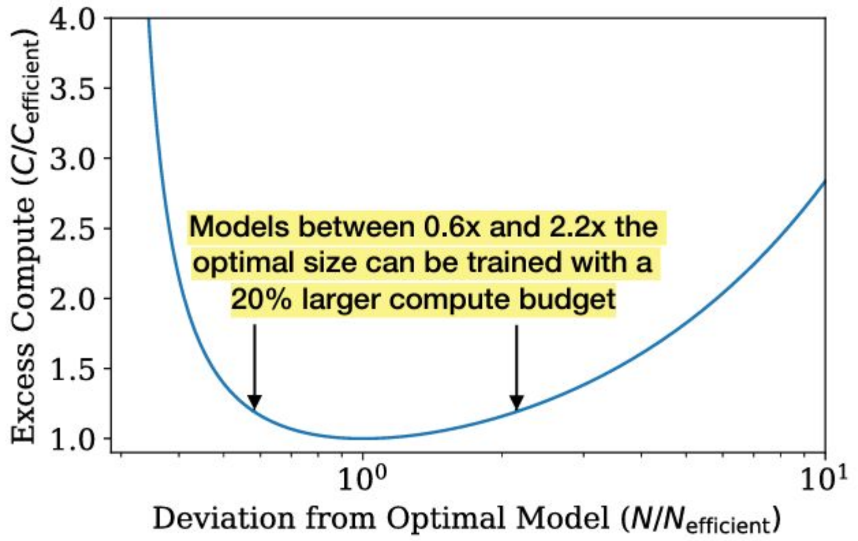


**Figure 4** **Left**: The early-stopped test loss $L(N, D)$ varies predictably with the dataset size $D$ and model size $N$ according to Equation (1.5). **Right**: After an initial transient period, learning curves for all model sizes $N$ can be fit with Equation (1.6), which is parameterized in terms of $S_{\min}$, the number of steps when training at large batch size (details in Section 5.1).



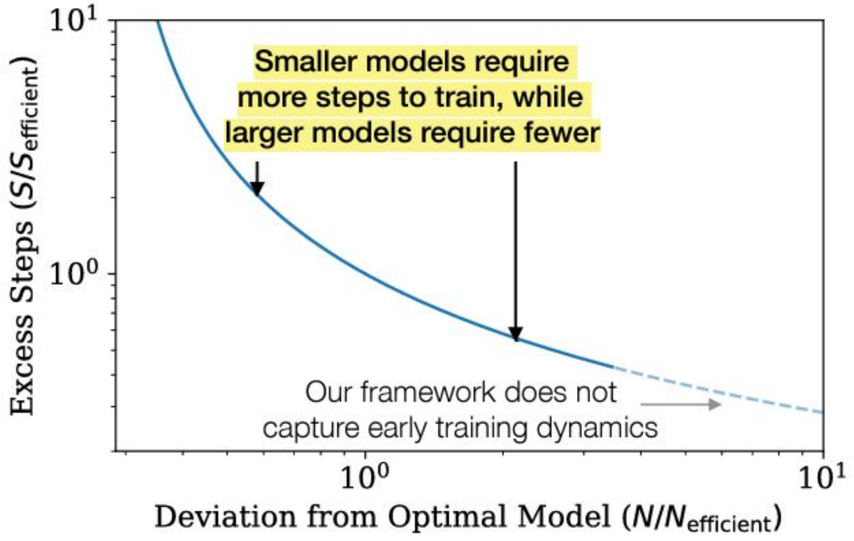


**Figure 12** **Left:** Given a fixed compute budget, a particular model size is optimal, though somewhat larger or smaller models can be trained with minimal additional compute. **Right:** Models larger than the compute-efficient size require fewer steps to train, allowing for potentially faster training if sufficient additional parallelism is possible. Note that this equation should not be trusted for very large models, as it is only valid in the power-law region of the learning curve, after initial transient effects.

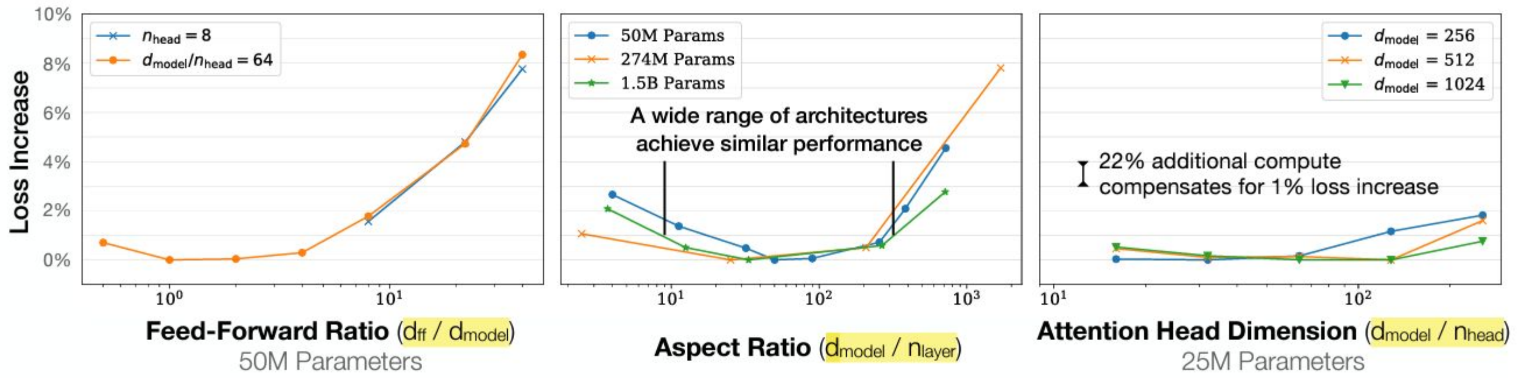
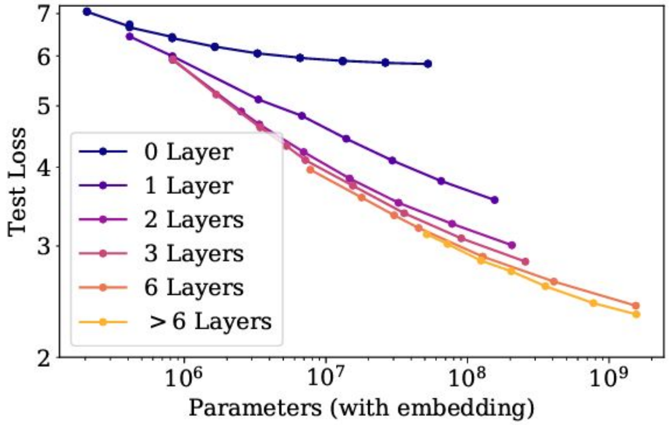


**Figure 5** Performance depends very mildly on model shape when the total number of non-embedding parameters $N$ is held fixed. The loss varies only a few percent over a wide range of shapes. Small differences in parameter counts are compensated for by using the fit to $L(N)$ as a baseline. Aspect ratio in particular can vary by a factor of 40 while only slightly impacting performance; an $(n_{\text{layer}}, d_{\text{model}}) = (6, 4288)$ reaches a loss within 3% of the $(48, 1600)$ model used in [RWC+19].



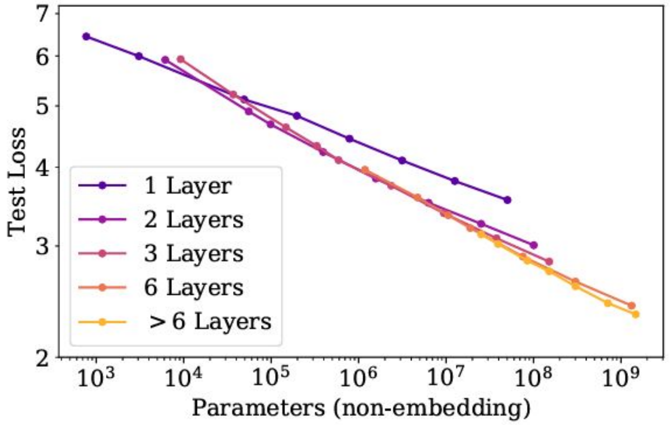
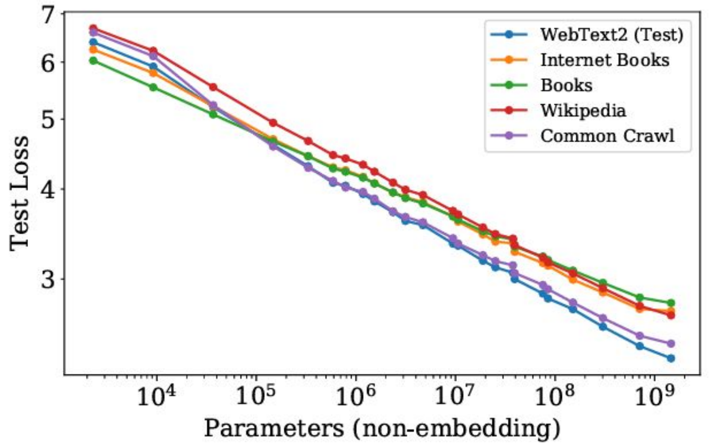


**Figure 6** **Left:** When we include embedding parameters, performance appears to depend strongly on the number of layers in addition to the number of parameters. **Right:** When we exclude embedding parameters, the performance of models with different depths converge to a single trend. Only models with fewer than 2 layers or with extreme depth-to-width ratios deviate significantly from the trend.

## Generalization



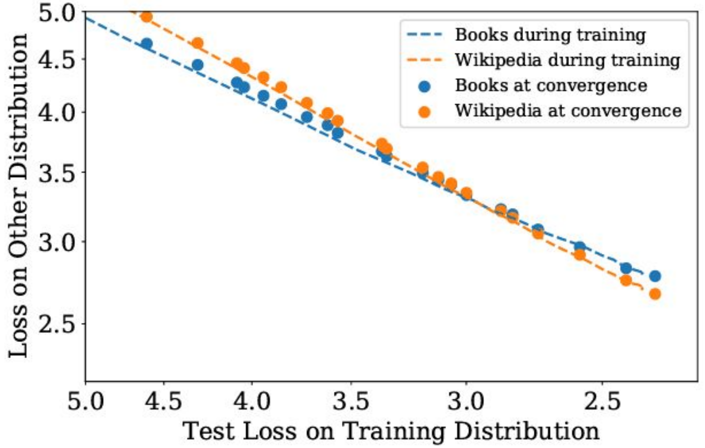


**Figure 8** **Left:** Generalization performance to other data distributions improves smoothly with model size, with only a small and very slowly growing offset from the WebText2 training distribution. **Right:** Generalization performance depends only on training distribution performance, and not on the phase of training. We compare generalization of converged models (points) to that of a single large model (dashed curves) as it trains.

# Critical batch size https://arxiv.org/pdf/1812.06162

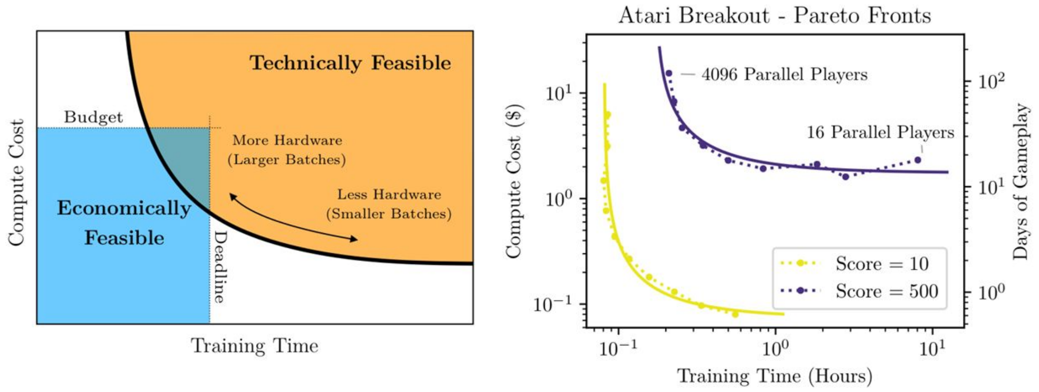


Figure 1: The tradeoff between time and compute resources spent to train a model to a given level of performance takes the form of a Pareto frontier (left). Training time and compute cost are primarily determined by the number of optimization steps and the number of training examples processed, respectively. We can train a model more quickly at the cost of using more compute resources. On the right we show a concrete example of the Pareto frontiers obtained from training a model to solve the Atari Breakout game to different levels of performance. The cost and training time depend on the computing architecture and are shown approximately.

# Critical Batch size

## 2.1 Intuitive Picture

Before working through the details of the gradient noise scale and the batch size, it is useful to present the intuitive picture. Suppose we have a function we wish to optimize via stochastic gradient descent (SGD). There is some underlying true optimization landscape, corresponding to the loss over the entire dataset (or, more abstractly, the loss over the distribution it is drawn from). When we perform an SGD update with a finite batch size, we're approximating the gradient to this true loss. How should we decide what batch size to use?

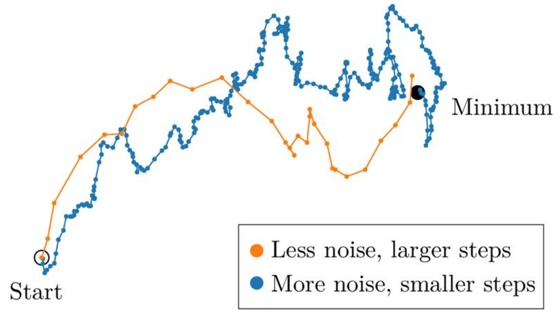


Figure 2: Less noisy gradient estimates allow SGD-type optimizers to take larger steps, leading to convergence in a smaller number of iterations. As an illustration, we show two optimization trajectories using momentum in a quadratic loss, with different step sizes and different amounts of artificial noise added to the gradient.

## Critical Batch size

When the batch size is very small, the approximation will have very high variance, and the resulting gradient update will be mostly noise. Applying a bunch of these SGD updates successively will average out the variance and push us overall in the right direction, but the individual updates to the parameters won’t be very helpful, and we could have done almost as well by aggregating these updates in parallel and applying them all at once (in other words, by using a larger batch size). For an illustrative comparison between large and small batch training, see Figure 2. By contrast, when the batch size is very large, the batch gradient will almost exactly match the true gradient, and correspondingly two randomly sampled batches will have almost the same gradient. As a result, doubling the batch size will barely improve the update– we will use twice as much computation for little gain. Intuitively, the transition between the first regime (where increasing the batch size leads to almost perfectly linear speedups) and the second regime (where increasing the batch size mostly wastes computation) should occur roughly where the noise and signal of the gradient are balanced– where the variance of the gradient is at the same scale as the gradient itself4. Formalizing this heuristic observation leads to the noise scale.

## Doesn’t answer

We train language models on WebText2, an extended version of the WebText [RWC+19] dataset, tokenized using byte-pair encoding [SHB15] with a vocabulary size nvocab = 50257. We optimize the autoregressive log-likelihood (i.e. cross-entropy loss) averaged over a 1024-token context, which is also our principal performance metric.

- All tests done with webtext2 - What about data quality - Phi
- Mostly Standard transformer models - (+LSTM)
- How does this impact downstream performance?
- Google-Chinchilla -

## Areas of additional interest

- Recurrent Transformers http://arxiv.org/abs/1807.03819
- Phi - Text books are all you need https://arxiv.org/pdf/2306.11644
  - Phi-3 https://arxiv.org/abs/2404.14219
- Google scaling laws - Training Compute-Optimal Large Language Models - Chinchilla - https://arxiv.org/abs/2203.15556

## Chinchilla

Whilst we reach the same conclusion, we estimate that large models should be trained for many more training tokens than recommended by the authors. Specifically, given a 10 increase computational budget, they suggests that the size of the model should increase 5□5 while the number of training tokens should only increase 1.8. Instead, we find that model size and the number of training tokens should be scaled in equal proportions.

# Google Chinchilla

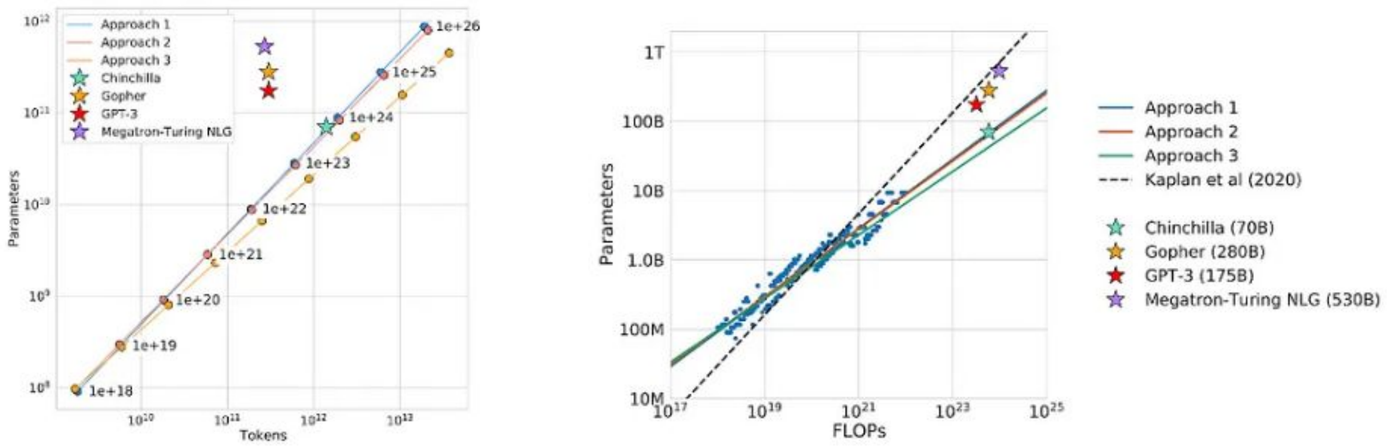


Fig4: DeepMind suggests a modified scaling law, indicating that models, including Gopher, GPT-3 and Megatron-Turing NLG can be trained with a lot less model parameters. (Left) Optimal number of tokens and parameters for a given FLOP budget (Right) Overlaid predictions of modified scaling law and OpenAI's scaling law

# Chinchilla laws

$$L(N, D) = E + \frac{A}{N^{0.34}} + \frac{B}{D^{0.28}}$$

Eq2: Parametric loss function fitted by DeepMind's scientists

## VS OpenAI

$$L(N, D) = \left[ \left( \frac{N_c}{N} \right)^{\frac{\alpha_N}{\alpha_D}} + \frac{D_c}{D} \right]^{\alpha_D}$$

| Parameter | $\alpha_N$ | $\alpha_D$ | $N_c$ | $D_c$ |
|---|---|---|---|---|
| Value | 0.076 | 0.103 | $6.4 \times 10^{13}$ | $1.8 \times 10^{13}$ |

**Table 2** Fits to $L(N, D)$

## References

https://medium.com/sage-ai/demystify-transformers-a-comprehensive-guide-to-scaling-laws-attention-mechanism-fine-tuning-fffb62fc2552