CSE 592: Foundations of AI — University of Michigan

Lecture 13: Scaling



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

Plan

1. Transformer architecture
2. Scaling laws
3. Generalization in deep learning
4. Systems at scale

Administrivia

In-class quiz today! Mostly for review, graded for completion (last ~30 min of class)

We will have another review next Wednesday

No class on Monday (Fall Study Break)

Where we left off: self-attention

Text → Token IDs (BPE tokenizer) → token embeddings (vectors)

The token vectors are the rows of the input matrix $X \in \mathbb{R}^{T \times d}$.

We add positional encoding: $X_i = E[x_i, :] + P_i$

Self-attention: for each token we compute a query, a key, and a value:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

Each position's output: an average of the values, weighted by how well its query matches each key.

$$O = \text{softmax}_{\text{rows}}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V$$

Causal mask M: position i only sees $j \leq i$. Otherwise, predicting the next token is trivial!

H heads, each with its own projections. Concatenate, then mix:

$$\text{MHA}(X) = \text{Concat}(O^{(1)}, \ldots, O^{(H)})W_O$$

So attention moves information between tokens. How do we build a whole neural network from this?

Finishing the Transformer

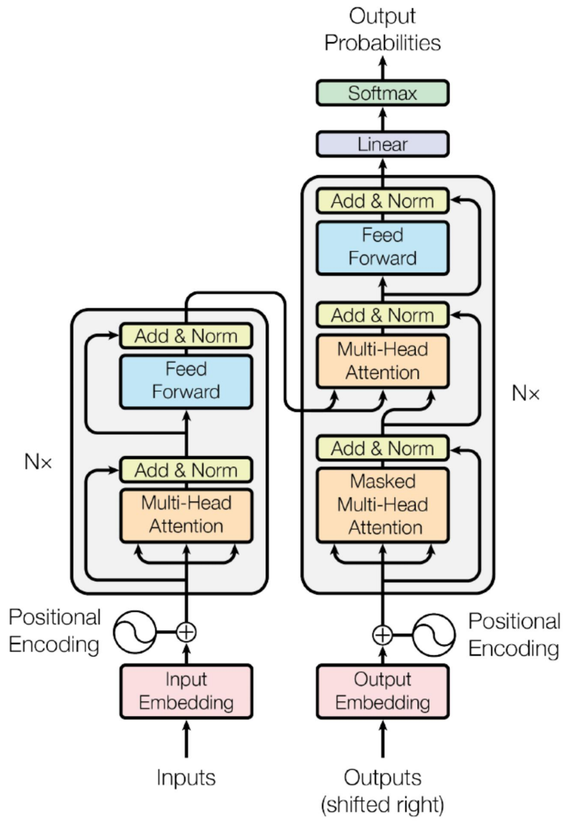
The original Transformer

Back to 2017: "Attention Is All You Need": a sequence-to-sequence model (for translation originally).

- **Encoder:** self-attention over the English sentence. No mask.
- **Decoder:** causal self-attention over the French written so far.
- **Cross-attention:** decoder queries, encoder keys and values.

Plus MLPs, residual connections, LayerNorm, and positions.

For a language model, we don't need an encoder: there's no separate input sentence.



Vaswani et al. (2017) — Fig. 1. Encoder at left, decoder at right.

A decoder-only language model

Keep only the decoder, without cross-attention.

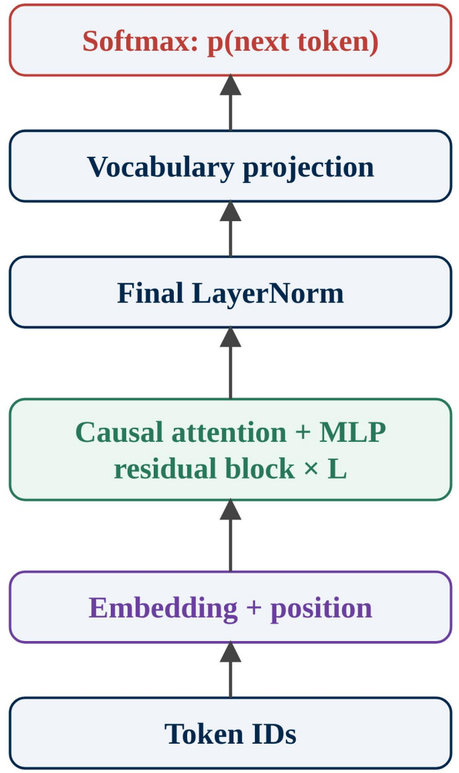
Embed the tokens and add positions.

Then a stack of L blocks. Each block has causal self-attention + an MLP.

At the top: logits over the vocabulary, at every position.

Train with cross-entropy loss on the next token.

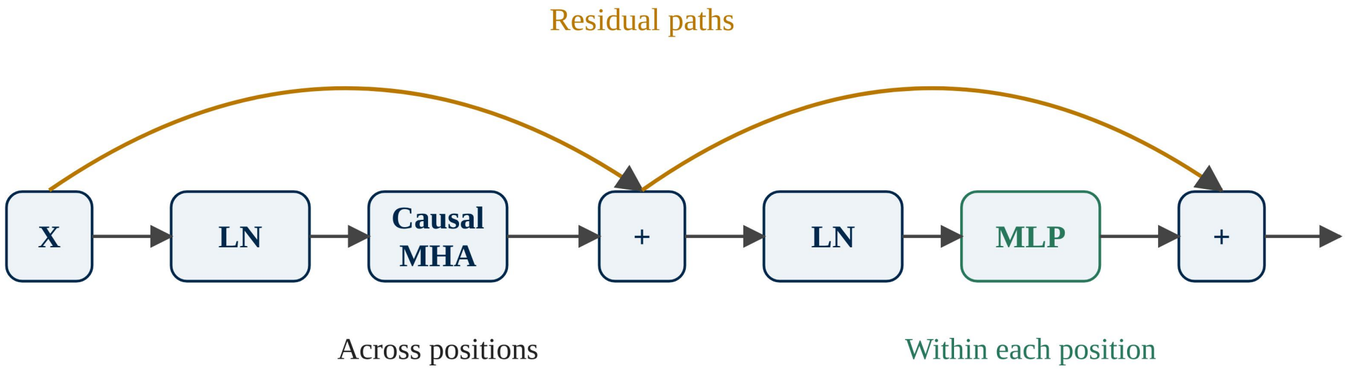
This is the architecture behind GPT.



One output distribution per input position

One Transformer block

Input $X \in \mathbb{R}^{T \times d}$. Output: same shape.



$$U = X + \mathrm{MHA}_{\mathrm{causal}}(\mathrm{LN}_1(X))$$

$$Y = U + \mathrm{MLP}(\mathrm{LN}_2(U))$$

Both tricks from lecture 11: residual connections, and LayerNorm (on each token's row).

Attention moves information *between* positions.

MLP transforms each position on its own: $\mathrm{MLP}(u) = \phi(uW_1 + b_1)W_2 + b_2$.

Same MLP at every position. Different MLPs in each of the L Transformer blocks.

This is a "pre-norm" Transformer, as more common now (at least since GPT-2; also see Xiong et al. (2020)).

The whole forward pass

Forward pass takes token IDs, outputs logits at each position.

```
X = token_embedding[ids] + position_embedding[:T]
for block in blocks:
    X = X + block.causal_attention(block.norm1(X))
    X = X + block.mlp(block.norm2(X))
X = final_norm(X)
logits = X @ W_out + b_out        # shape: T × V
```

(Batch dimension omitted here - we typically take a Tensor of shape $T \times B \times V$)

Every line is differentiable: all of it is trained together, by backprop.

The training loop

A batch of token sequences. Shift them by one to get inputs and targets:

```
inputs  = tokens[:, :-1]
targets = tokens[:, 1:]
logits  = model(inputs)           # B × T × V
loss    = cross_entropy(logits, targets)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Same loss as for the RNN. Only the architecture changed.

Optimizer: one common choice is AdamW, with warmup and cosine decay.

Parallel training, sequential generation

Training: we know all the input tokens.

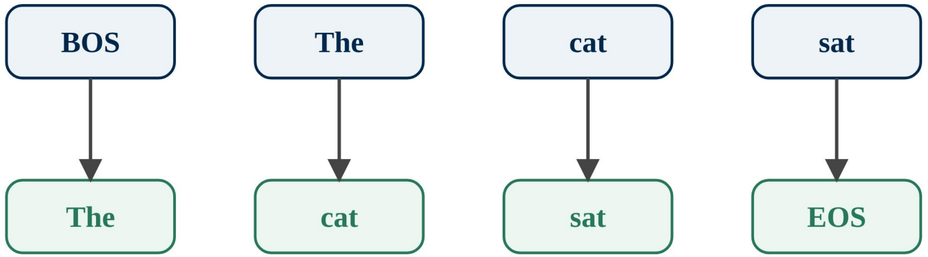
Compute all positions **in parallel (!)**: a few big matrix multiplications.

> 🧠 **Recall**
>
> Recall: an RNN has to go one token at a time, even in training. Transformers make much better use of GPU parallelism.

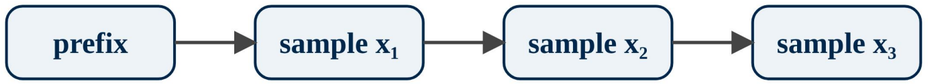
The mask makes sure each prediction can't use later tokens.

Generation: the next token isn't known until we sample it.

Still one token at a time: sample, append, repeat.



Training: all targets are already available



Generation: each sampled token extends the prefix

GPT: pretraining on unlabeled text

Now we have the whole recipe: architecture, loss, training loop. What can we learn from text alone?

Problem: most tasks (e.g., sentiment classification) have few labeled examples. But there's lots of unlabeled text.

Idea (GPT, 2018): first train a language model on lots of text (**pretraining**).

Then adapt it to each task with its labeled data (**fine-tuning**).

Model: 12 decoder blocks, 117M parameters, context of 512 tokens.

Data: BooksCorpus, over 7,000 books. Long, contiguous text: good for learning long-range dependencies.

$$\min_{\theta} \; -\sum_{t} \log p_\theta(x_t \mid x_{<t})$$

The same next-token loss we just implemented. No task labels yet.

Now how do we perform a particular prediction task (e.g., entailment)?

We can "fine-tune" it with a little bit of task-specific training

Radford et al., Improving Language Understanding by Generative Pre-Training (2018)

GPT: fine-tuning on downstream tasks

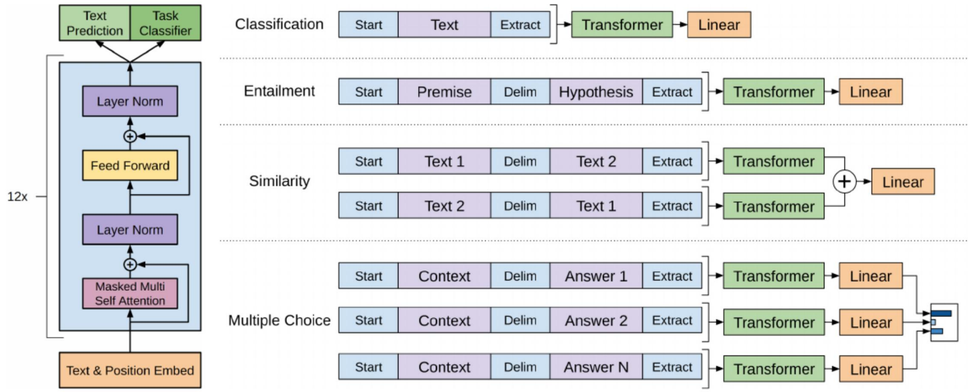


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Radford et al., Improving Language Understanding by Generative Pre-Training (2018) —

Idea: Write each task-specific input as a token sequence. E.g., *entailment* (does a premise imply a hypothesis?): premise + delimiter + hypothesis.

Add a linear classifier on the last token's vector h_{last}:

$$p(y \mid x) = \text{softmax}(h_{\text{last}} W_y)$$

Update **all** the weights on the labeled examples. One fine-tuned model per task.

(They also keep the next-token loss as an extra, “auxiliary” loss while fine-tuning.)

Did pretraining actually help?

Control: the same Transformer, trained only on each task's labeled data (i.e., no pre-training).

Method	Avg. Score	CoLA (mc)	SST2 (acc)	MRPC (F1)	STSB (pc)	QQP (F1)	MNLI (acc)	QNLI (acc)	RTE (acc)
Transformer w/ aux LM (full)	74.7	45.4	91.3	82.3	82.0	**70.3**	**81.8**	**88.1**	**56.0**
Transformer w/o pre-training	59.9	18.9	84.0	79.4	30.9	65.5	75.7	71.2	53.8
Transformer w/o aux LM	**75.0**	**47.9**	**92.0**	**84.9**	**83.2**	69.8	81.1	86.9	54.4
LSTM w/ aux LM	69.1	30.3	90.5	83.2	71.8	68.1	73.7	81.1	54.6

Radford et al. (2018) — Table 5. One column per task, each with its own metric.

First two rows, without → with pretraining. Sentiment classification (SST-2): 84.0 → 91.3% accuracy.

Entailment (MNLI): 75.7 → 81.8% accuracy.

Same labeled examples! Pretraining changes what the model can learn from them.

But we still need labeled data, and a separate fine-tuned model for every task...

GPT-2: same recipe, bigger

2019: same architecture and loss, with small tweaks (e.g., pre-norm instead of post-norm).

More data: WebText, about 40GB of text from web pages.

Bigger models. Largest: 1.5 billion parameters, 48 blocks.

New: evaluate on tasks **without fine-tuning**. Write a prompt, and let the model continue it.

E.g., a passage + a question + “Answer:” + sample continuation from the model.

No labeled examples, no weight updates: **zero-shot**.

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

Radford et al., Language Models are Unsupervised Multitask Learners (2019) — Table 2.

GPT-2: a story about unicorns

The authors wrote the start of a (fake) news story:

Context (human-written): In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

GPT-2 continues:

GPT-2: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

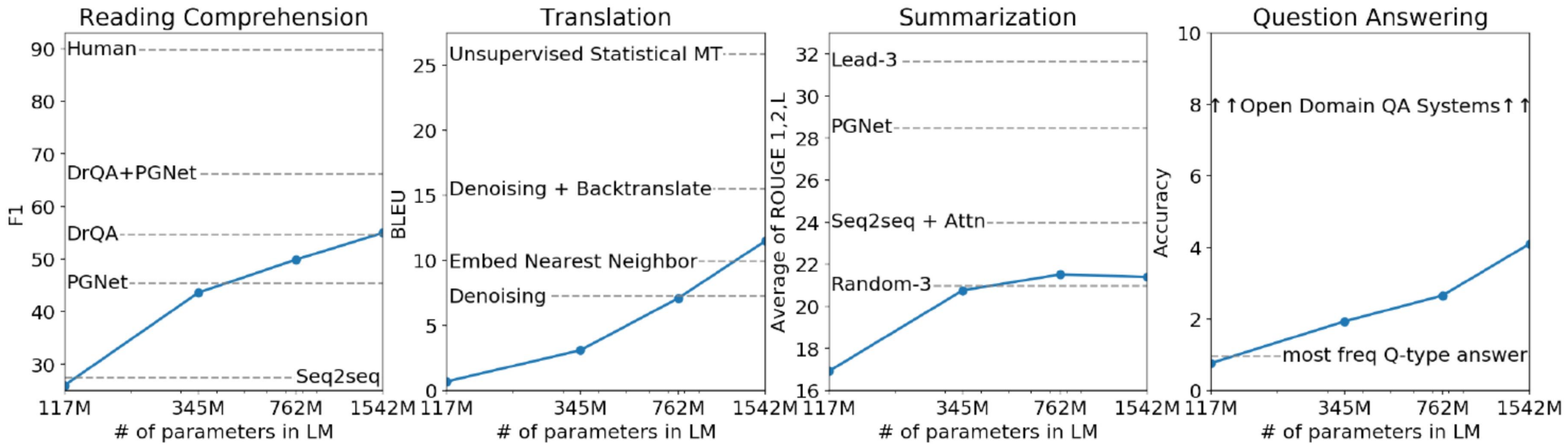
Radford et al. (2019) — Table 13, from the opening of the story. The authors' pick out of 10 samples.

Stays on topic, in the style of a news report, for several paragraphs.

Not perfect: "their distinctive horn", then "four-horned" in the next sentence!

GPT-2: bigger models, better zero-shot

Does GPT-2 work zero-shot?



Radford et al., Language Models are Unsupervised Multitask Learners (2019) — Fig. 1. Each panel: a different task, with its own metric.

No fine-tuning on any of these tasks. Only next-token prediction on web text.

Generally: bigger model, better zero-shot performance. Still far from the best systems on most tasks.

How far can this go?

Each new GPT: roughly 10–100× bigger than the last.

	GPT (2018)	GPT-2 (2019)	GPT-3 (2020)
Parameters	117M	1.5B	175B
Transformer Blocks	12	48	96
Max context length	512	1,024	2,048
Evaluation	Fine-tune	Zero-shot	Zero / few-shot

Same core: decoder-only Transformer + next-token loss.

GPT-3: put a few solved examples in the prompt (**few-shot**). Still no weight updates!

The model picks up the task from its input: **in-context learning**.

> **? Question**
>
> Can we just keep making models bigger? Will they improve?
> How much data and compute will that take? How much better will they get?

Radford et al. (2018); Radford et al. (2019); Brown et al. (2020).

Scaling laws

What are we scaling?

To answer this, we need to measure “better” and “bigger”.

“Better”: lower next-token loss $\mathcal{L}$ on **held-out text** (the same cross-entropy as before).

“Bigger”: three resources we can increase.

- N: number of parameters.
- D: number of training tokens.
- C: training compute, in floating-point operations (FLOPs).

They're related: approx. 6 FLOPs per parameter, per training token (forward + backward).

$$C \approx 6ND$$

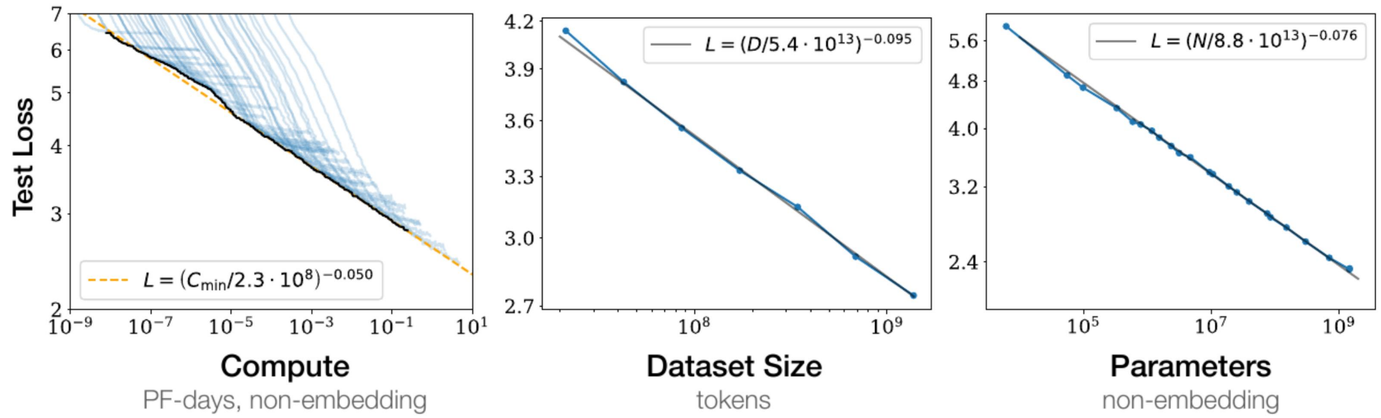
Kaplan et al. (2020), Formula from Sec 2.1 - just a rough count for a dense Transformer, ignoring, e.g., the attention scores at very long contexts.

Q: With a fixed compute budget, can we double N and keep D the same?

A: No! Twice the parameters means half as many training tokens. We have to choose how to split C.

Kaplan et al.: smooth power laws

2020: Kaplan et al. at OpenAI trained many Transformer LMs, of many sizes, on different amounts of data, measured loss on held-out text data.



Kaplan et al., Scaling Laws for Neural Language Models (2020) — Fig. 1. Log–log axes. In each plot, the other two resources aren't the bottleneck.

One plot per resource. Log–log axes, and the points fall on straight lines: **power laws**.

$$\mathcal{L}(N) \propto N^{-0.076}, \qquad \mathcal{L}(D) \propto D^{-0.095}$$

Q: What happens to the loss if we double the parameters?

A: It gets multiplied by $2^{-0.076} \approx 0.95$: about 5% lower.

Small gains, but very predictable, over many orders of magnitude.

Bigger models use data better

Each curve: one model size. Horizontal axis: dataset size.

Q: With the **same** data, which model reaches lower loss?

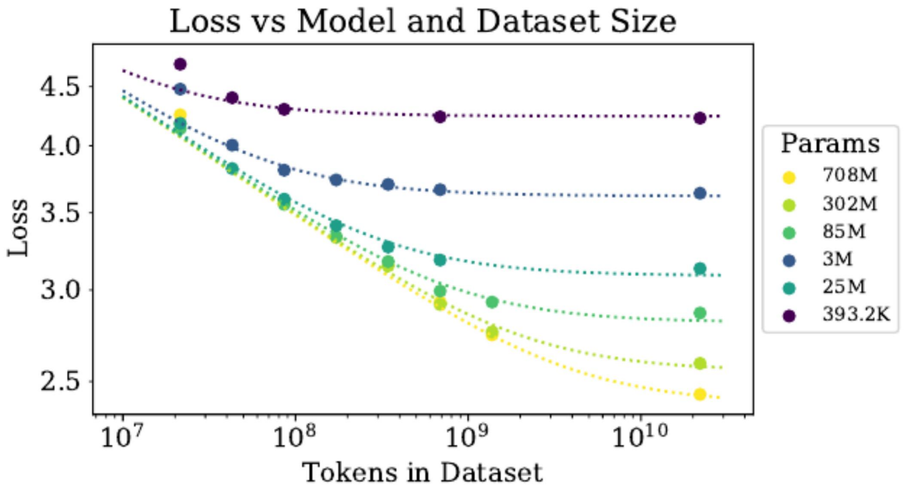
A: The bigger one!

Or: the same loss with fewer tokens. Bigger models are more **sample-efficient**.

But each curve flattens: eventually, model size is the bottleneck.

With little data, size barely matters: data is the bottleneck.

Note: each run stops when test loss stops improving **(early stopping)**. Training longer on little data, models still overfit (test loss goes up).



Kaplan et al. (2020) — Fig. 4, left. Each point: test loss with early stopping.

Kaplan: how should we spend more compute?

Given a budget C, how big a model should we train on how many tokens for best test loss?

A small model trained to convergence gets stuck: limited by its size.

A larger model, trained for fewer steps (before it converges), can reach a lower loss with the same compute.

Their fit for the best allocation:

$$N_{\text{opt}} \propto C^{0.73}, \qquad D_{\text{opt}} \propto C^{0.27}$$

10x compute $\rightarrow$ about 5.4x parameters, but only 1.9x training tokens.

Recommendation: spend most of the extra compute on model size.

The big models that followed did just that: GPT-3 (175B) and Gopher (280B), both trained on ~300B tokens.

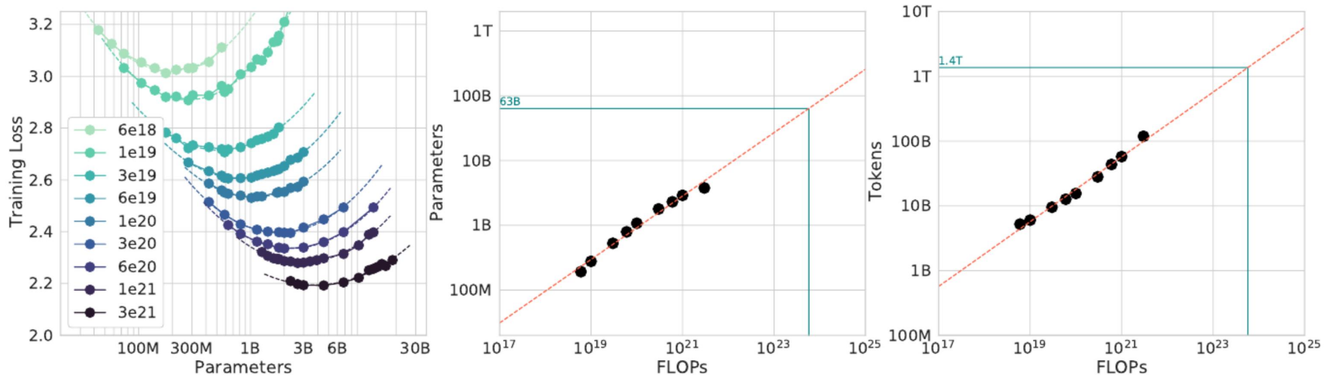
But this is an empirical fit. Does it extrapolate well to larger scales?

Kaplan et al. (2020), §6 and Appendix B. Token counts from Hoffmann et al. (2022), Table 1.

Chinchilla: scale size and data together

2022: Hoffmann et al. revisit the question (Google's "Chichilla Scaling Laws" paper).

Over 400 models, fixing C (color), varying size (x) *and* number of training tokens (y).



Hoffmann et al., Training Compute-Optimal Large Language Models (2022) — Fig. 3. Left: one curve per compute budget. Center / right: best model size / number of tokens for each budget.

Left: each curve fixes the compute budget, and varies model size.

Q: Why does each curve have a minimum?

A: Too small: not enough capacity. Too large: too few tokens to train it.

Center and right: the best size and number of tokens, for each budget.

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

4x compute $\rightarrow$ 2x parameters **and** 2x tokens. Scale both equally.

Chinchilla: testing the predictions

Tested the prediction on Gopher's compute budget (Google's 280B LLM from 2021). What's the best model for it?

Prediction: about 70B parameters, on 1.4T tokens. They trained it, and called it Chinchilla.

	Gopher	Chinchilla
Parameters	280B	70B
Training tokens	300B	1.4T
MMLU accuracy (exam questions, 5-shot)	60.0%	67.6%

Same training compute. Chinchilla is better on most downstream tasks.

Also cheaper to run: 4x fewer parameters, for every token we generate!

Why did Kaplan et al. get a different answer? One reason: Kaplan used same learning rate schedule for every run, whatever its length. (Schedules matter!)

Rule of thumb from these experiments: about 20 training tokens per parameter.

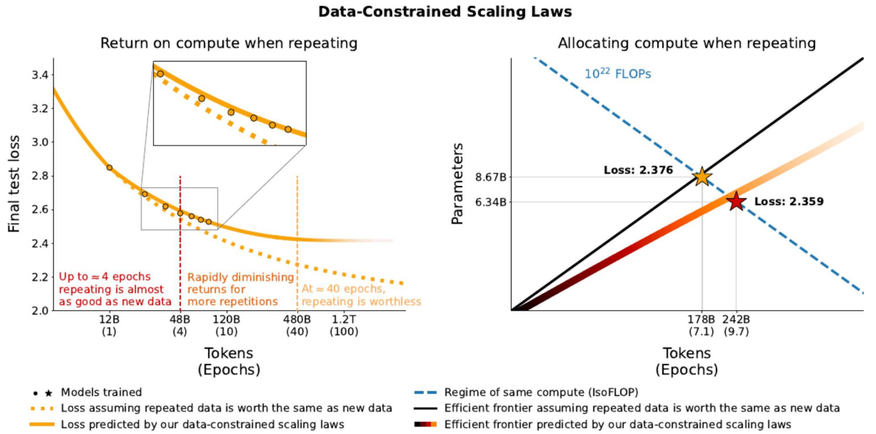
Hoffmann et al. (2022), Secs 2–4, Tables 1 and 6. Chinchilla also changed other details of training and data: not a clean experiment on size alone.

What if we run out of fresh text?

So far, we assumed we can always get more new text data.

At 20 tokens per parameter, a 1T-parameter model wants 20T tokens!

Option: repeat the data, i.e., train for several **epochs**.



Muennighoff et al., Scaling Data-Constrained Language Models (2023) — Fig. 1. Left: a 4.2B model, trained for increasingly more epochs. Right: best allocation of a fixed budget, when repeating.

Up to about 4 epochs: almost as good as fresh data!

After that: diminishing returns. Around 40 epochs, more repetition barely helps.

Right: with repeated data, a smaller model trained for more epochs does better.

Repeated tokens are worth less than fresh ones. Scaling laws need to account for that.

What do scaling laws tell us?

In classical ML, we trained many models to pick the best one.

Now, training is too expensive to even do twice (e.g., GPT-4 pre-training run was $100M).

> 💡 **Insight**
> Scaling laws show that LLM pretraining is surprisingly predictable!

Meaning: we can predict the test loss of huge run from much smaller runs

E.g., GPT-4's final loss was predicted from runs with 10,000x less compute.

But: each fit is highly specific to one setup.

We care about performance on tasks, not perplexity (harder to predict).

Also we only counted *training* (not inference after deployed)

> ? **Question**
> Still: bigger models kept getting better, over many orders of magnitude.
> Didn't we say there was a “sweet spot” in model size?

OpenAI, GPT-4 Technical Report (2023) and Touvron et al., LLaMA (2023)

Generalization in deep learning

Back to generalization

> 🧠 **Recall**
> Lecture 9: a larger hypothesis class $\mathcal{H}$ has less approximation error, but potentially more estimation error.

So test error should be U-shaped: a “sweet spot” in model size.

PAC bound: the training data we need grows with $\mathcal{H}$.

But we just saw bigger language models do better *with the same data*.

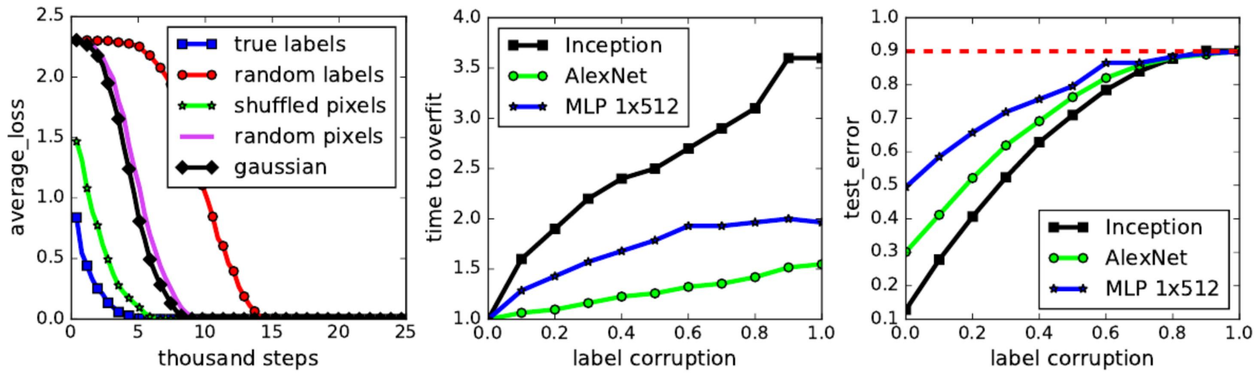
And image classifiers often have more parameters than training images (e.g., ResNet-18: 11M parameters, 50k images in CIFAR-10).

Maybe neural networks are effectively a small $\mathcal{H}$, somehow?

We can test it: can a deep network fit arbitrary labels (get to 0 loss)?

Experiment (Zhang et al., 2017): take CIFAR-10 images, but replace the true labels with random ones.

Neural nets can fit random labels!



(a) learning curves (b) convergence slowdown (c) generalization error growth

Zhang et al., Understanding Deep Learning Requires Rethinking Generalization (2017) — Fig. 1, on CIFAR-10. Training loss (left), time to fit (middle), test error (right).

Left: training loss goes to 0, even with random labels! Just takes a bit longer.

Model memorizes 50k arbitrary labels. So $\mathcal{H}$ is not small.

Right: test error, as we randomize more and more of the labels.

Real labels: it generalizes. All random labels: 90% error, i.e., random predictions (10 classes)

Same networks, same training. So why do they learn something useful from real labels?

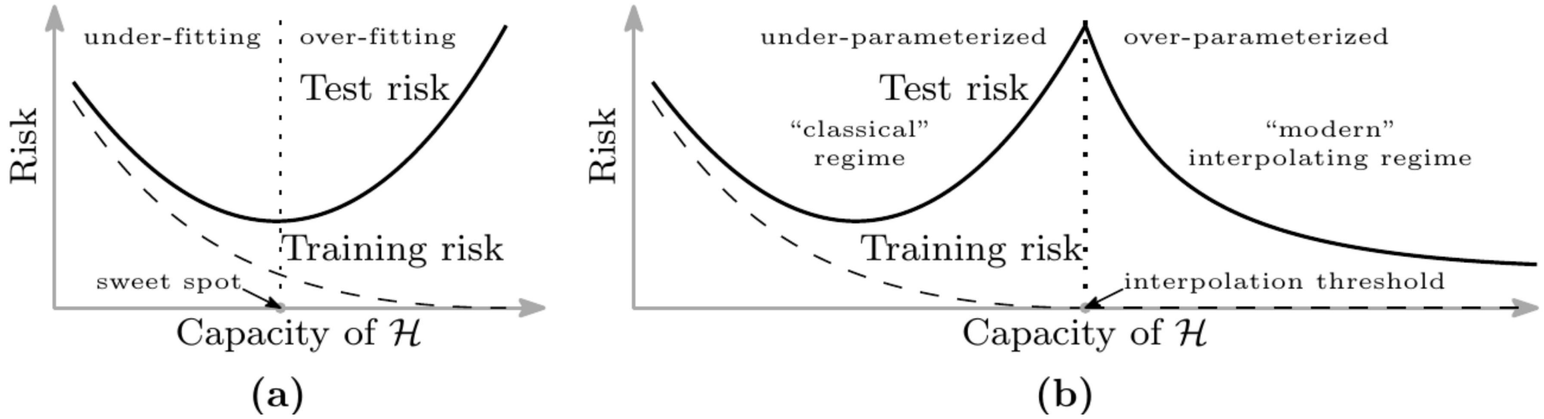
What if we keep increasing model size?

Say we fix the training data, increase parameters, train, and plot test error.

At first, test error goes down: more expressive models. Then goes up: we start to overfit.

Worst at the **interpolation threshold**: just enough to memorize training set.

In classical ML, we used to stop here! What if we keep going?



Belkin et al., Reconciling Modern Machine Learning Practice and the Bias–Variance Trade-off (2019) — Fig. 1. Schematic. (a) The classical picture. (b) Past the interpolation threshold.

Test error falls again! This is **double descent**, a somewhat recently observed phenomenon.

In fact, this even happens with linear regression. The theory is already pretty complete in that case.

Hastie, Montanari, Rosset & Tibshirani, Surprises in High-Dimensional Ridgeless Least Squares Interpolation.

Systems at scale

Can one GPU train Chinchilla?

Scaling laws say: go big. Say we want to train Chinchilla: 70B parameters, on 1.4T tokens.

A modern GPU (e.g., NVIDIA H100):

- **Memory:** 80 GB.
- **Speed:** about 10^{15} floating-point operations per second (FLOP/s).

Recall: training compute $C \approx 6ND$.

$$C \approx 6 \times (70 \times 10^9) \times (1.4 \times 10^{12}) \approx 6 \times 10^{23} \text{ FLOPs}$$

One GPU, at full speed: $6 \times 10^{23} / 10^{15} = 6 \times 10^8$ seconds. About 19 years!

So: we need many GPUs to parallelize training. E.g., Llama 3 (405B params, 2024) used up to 16k GPUs.

Memory next: does the model even fit?

First, how many bytes does each parameter take?

10^{15} FLOP/s is the H100's peak speed for 16-bit arithmetic. Real training runs reach maybe 40% of this peak. Llama Team, The Llama 3 Herd of Models (2024).

How do we store real numbers?

Weights, gradients, activations are all floats taking some number of bits.

Idea (**floating point**): think of it in scientific notation. E.g., 6.02×10^{23}.

The digits give the *precision*. The exponent gives the *range*.

In binary: one **sign** bit, some bits for the **exponent**, and the rest for the digits (the **mantissa**).

Format	Bits	Digits	Largest value
float32	8 | 23	~7	3×10^{38}
float16	5 | 10	~3	65,504
bfloat16	8 | 7	~2	3×10^{38}

32 bits: the usual default. Large models mostly compute in 16 bits (or even fewer).

Half the memory, and much faster arithmetic on modern GPUs.

Q: What do we lose?

A: float16: small range. Large values overflow to ∞; tiny gradients become 0.

bfloat16 (“brain float”, from Google Brain): same range as float32, but even fewer digits.

Small updates get rounded away

Fewer digits: representable numbers are farther apart.

In bfloat16, the numbers right after 1 are 1.0078, 1.0156, ... (steps of 2^{-7}).

Q: Say a weight is $w = 1$. Gradient step changes it by 0.001. What's the new w, in bfloat16?

A: Rounds back to 1. No update!

In float32, the spacing near 1 is $2^{-23} \approx 10^{-7}$. The update would happen.

Idea (**mixed precision**): keep a float32 “master copy” of the weights, and apply the updates there.

Use 16-bit copies for the forward and backward passes, where almost all the arithmetic happens.

Micikevicius et al., Mixed Precision Training (2018)

Fancier optimizer need more memory

Memory again: What do we store while training a 70B model?

> 🧠 **Recall**
> Adam keeps two running averages: u (of the gradients, momentum) and v (of the squared gradients, like RMSProp).

Q: How many floats are in u? And in v?

A: One per parameter! Each is as big as the model itself!

Plain SGD: weights + gradient. Momentum adds u. Adam adds u and v on top: 2x the memory of SGD.

Mixed precision + Adam	Bytes / parameter
16-bit weights + gradients (forward and backward)	2 + 2
float32 master copy of the weights	4
float32 u and v	4 + 4
Total	**16**

$70 \times 10^9 \times 16$ bytes ≈ 1.1 TB. That's the memory of 14 GPUs, just for the model state!

Rajbhandari et al., ZeRO (2020)

And then the activations

> 🧠 **Recall**
> When training, we need to keep the activations from the forward pass, for backprop.

In a Transformer: every block stores its intermediate vectors, for every token.

So this grows with (number of blocks) × (width) × (tokens in the batch).

Chinchilla: 80 blocks, width 8,192. Rough estimate: ~45 GB for a single 2,048-token sequence!

And a batch has hundreds of sequences. Much more than the 1.1 TB of model state.

Idea (**activation checkpointing**): only store the input of each block. During backprop, run the block's forward pass again to get the rest.

Trades compute for memory: costs about one extra forward pass.

Estimate: about 34 bytes per token, per block, per unit of width, in 16 bits, without the attention scores (Korthikanti et al., 2022). Checkpointing: Chen et al., Training Deep Nets with Sublinear Memory Cost (2016).

Many GPUs: split the data

So we need many GPUs: for speed (19 years!) and for memory (1.1 TB, plus activations). How do they share the work?

Recall: the minibatch gradient is an average over examples.

Idea (**data parallelism**): each GPU holds a copy of the model, and gets different examples from the batch.

Each GPU computes the gradient on its examples. Then average across GPUs, and every GPU applies the same update.

Q: Does this solve our memory problem?

A: For activations, yes: each GPU only stores those of its own examples. But every GPU still holds all 1.1 TB of model state!

Idea (**sharding**): also split the weights, gradients, and optimizer state. Each GPU keeps a slice, and fetches the rest from the others when it needs them.

New cost: communication. At every step, GPUs send each other gradients and weights.

Rajbhandari et al., ZeRO (2020): sharding for data parallelism.

Many GPUs: split the model

Another option: give each GPU part of the *model*, instead of part of the data.

Idea (**pipeline parallelism**): GPU 1 runs blocks 1–20, GPU 2 runs blocks 21–40, and so on.

Problem: GPU 2 waits for GPU 1's outputs. Fix: cut the batch into smaller pieces, so the GPUs can work on different pieces at once.

Idea (**tensor parallelism**): split the matrices themselves. E.g., each GPU computes some of the columns of XW.

But then GPUs exchange results inside every block. Only fast within one machine.

Method	What goes on different GPUs?	What must move?
Data parallelism	Different examples	Gradients (and weights, if sharded)
Pipeline parallelism	Different groups of blocks	Activations and their gradients
Tensor parallelism	Pieces of each matrix	Partial results, inside each block

Big runs combine all three. GPUs often sit idle, waiting for data from other GPUs.

Runs take weeks, and hardware fails. Save checkpoints often, so a crash doesn't send us back to step 1.

Narayanan et al., Efficient Large-Scale Language Model Training on GPU Clusters (2021).

The cost of a long context

Most costs so far grow linearly with the number of tokens. But attention compares *pairs* of tokens.

Recall: every query is compared with every (allowed) key. Context of T tokens: T^2 scores, for each head, in each block.

Double the context $\rightarrow$ 4× the attention work. Plus a $T \times T$ score matrix to store.

Idea (**FlashAttention**): compute attention in small tiles, and never store the whole matrix.

Same result, less memory, faster. But still T^2 arithmetic.

Generation: one token at a time. Naively, each new token reruns the model on the whole prefix.

But the keys and values of earlier tokens never change: they can't see later tokens. Store and reuse them: the **KV cache**.

Problem: the cache grows with the context, and with the number of users. Memory, again!

Dao et al., FlashAttention (2022); Vaswani et al. (2017), Table 1.

More parameters, without using all of them

So far, every token goes through every parameter (a **dense** model).

Q: Can we add parameters, without adding compute per token?

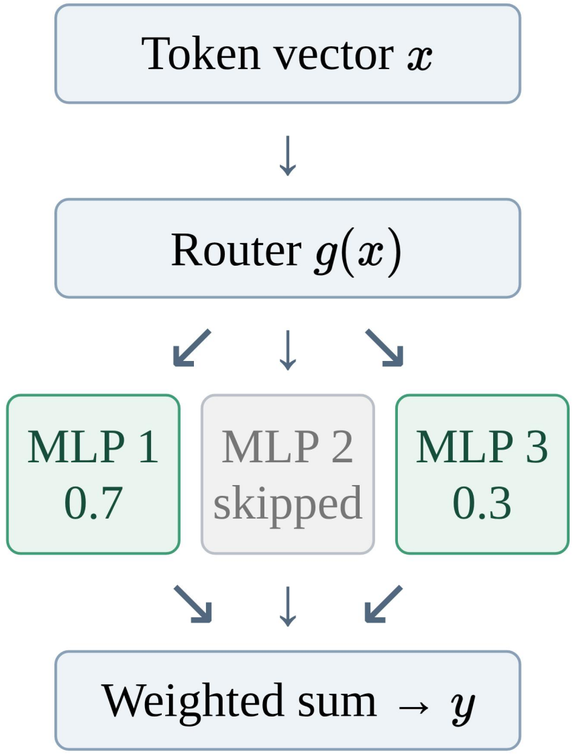
Idea (**mixture of experts**, MoE): replace each MLP with several MLPs, the “experts”.

A learned router picks a few of them for each token:

$$y = \sum_{e \in \mathrm{TopK}(g(x))} g_e(x)\,\mathrm{MLP}_e(x)$$

More experts → more parameters, with the same compute per token.

Catch: we still store all the experts (potentially in different GPUs).



Example: top-2 routing.

Fedus, Zoph & Shazeer, Switch Transformers (2021/2022)

All of this is still pretraining

Everything today: train to predict likely text continuations. That gets quite far!

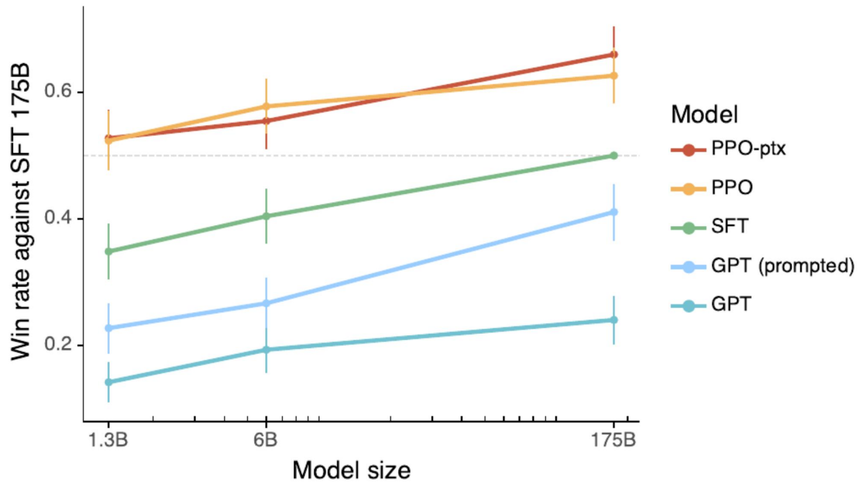
But likely continuations isn't always what we want.

E.g.: if you prompt with a question, the likely continuation might be... other similar questions (happens on many web pages).

Post-training: takes pre-trained model and adapts it to follow instructions, answer questions, generate reasoning traces, etc...

InstructGPT (2022): after post-training, people preferred a 1.3B model's answers to the 175B GPT-3's!

We'll see how this works later, after reinforcement learning.



Ouyang et al., Training Language Models to Follow Instructions with Human Feedback (2022) — Fig. 1. How often people preferred each model's answer, against a fixed 175B baseline. GPT: no post-training.

Summary

- **Transformer:** causal attention + MLPs + residuals + LayerNorm, in parallel over positions.
- **GPT models:** pretraining on text learns useful features. At scale, a task can be just posed in a prompt.
- **Scaling laws:** loss falls predictably with parameters, data, and compute. Scale size and data together.
- **Generalization:** past the interpolation threshold, bigger can be better. Which solution training finds matters.
- **Systems: scaling is hard!** Memory (precision, optimizer state, activations) and communication between GPUs are training bottlenecks.

Midterm covers up to here!