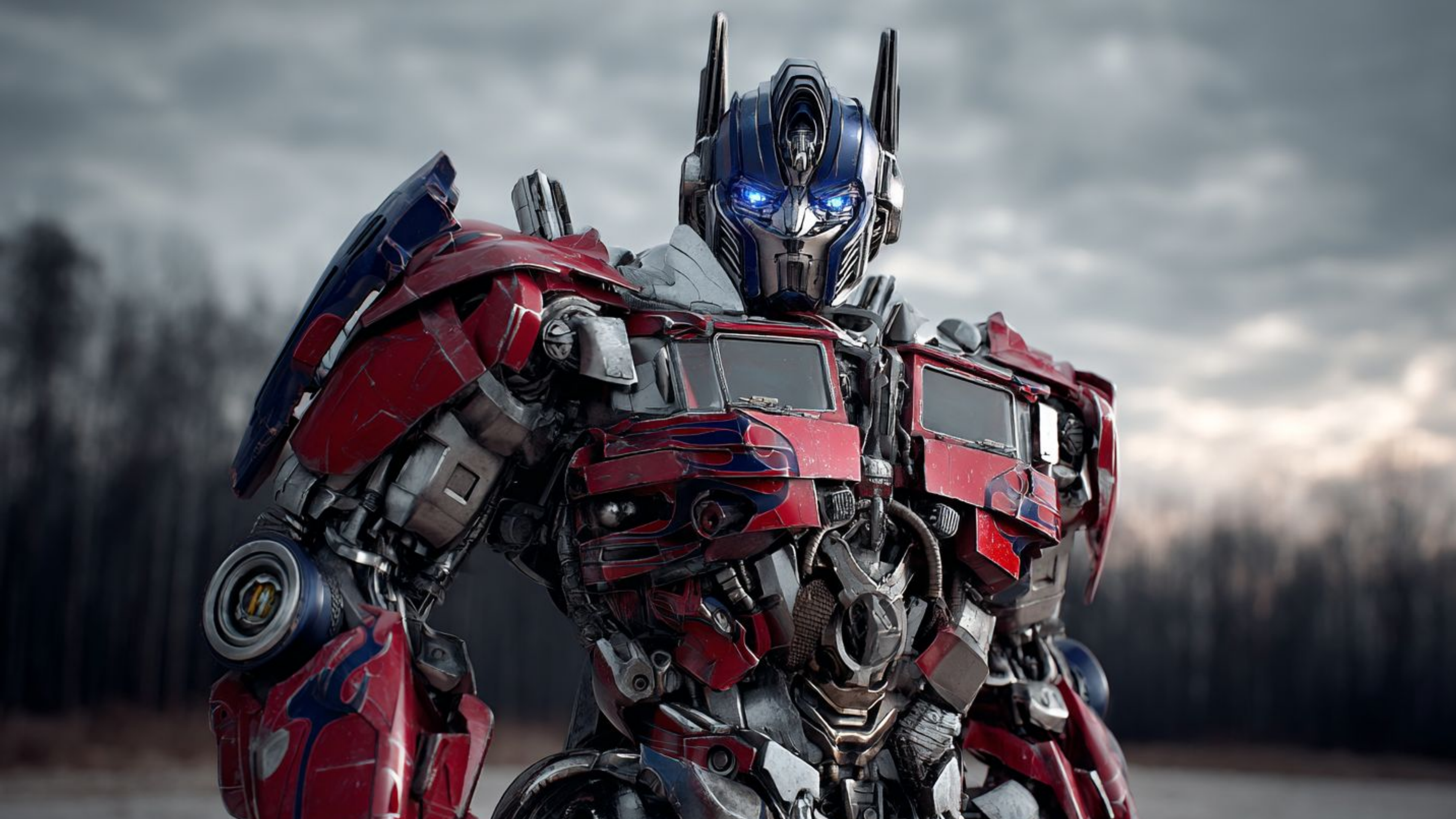
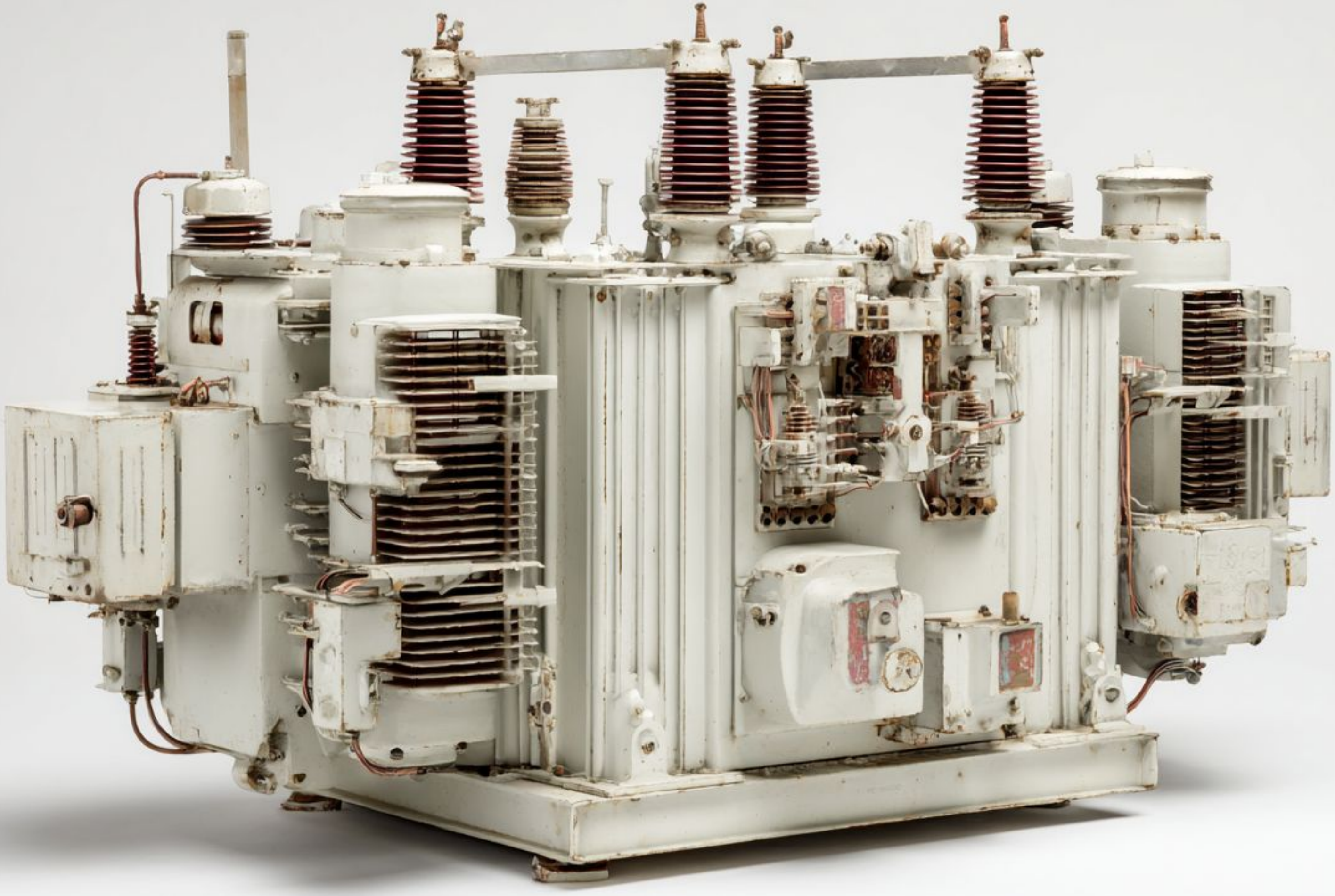
Transformers & Generative AI

Finbarr Timbers

✉ finbarrtimbers@gmail.com
𝕏 @finbarrtimbers
🌐 https://finbarr.ca/transformers





This transformer.

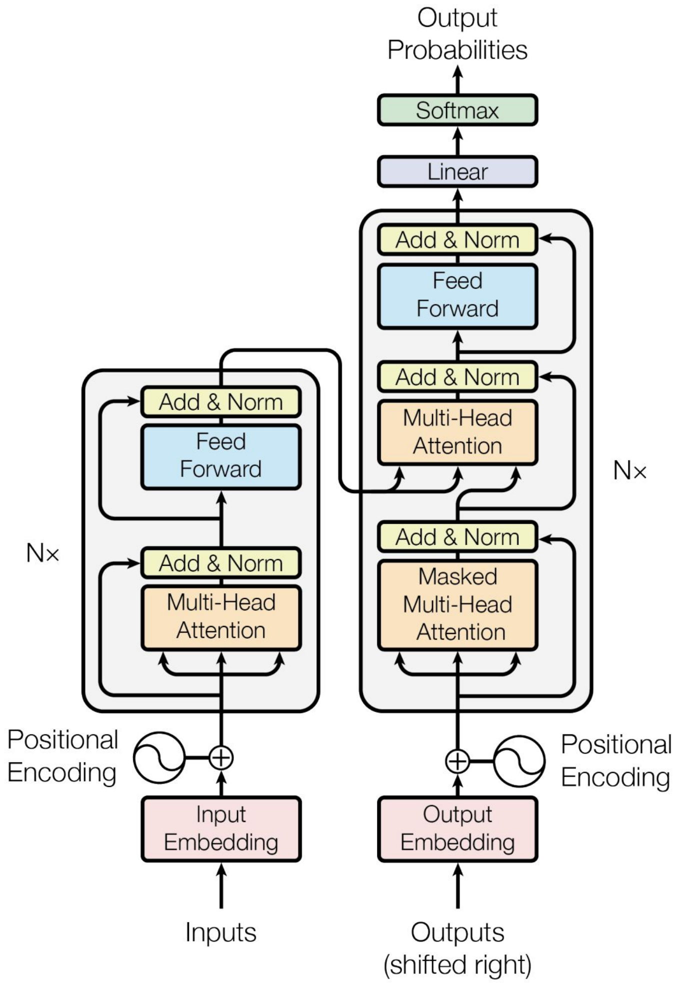


Figure 1: The Transformer - model architecture.

Transformers

- Incredibly useful architecture introduced by Vaswani et. al in 2017
- Increasingly dominate ~all ML
    - Certainly by % of FLOPS used in production
- Goal today is to discuss why they work and develop some intuition
- Also discuss why generative AI works

Generative AI

Two major events in 2022, which sparked the generative AI explosion:

1. Stable Diffusion, released in August
2. ChatGPT, released in November

What made generative AI possible?

1. The transformer.
2. Pretraining.

Pretraining

- Imagine the world of ML in ye olde 2014
- Supervised learning dominates

Pretraining

- To solve a new problem, you need new data
- Consider what this looks like: you spend most of your compute teaching the model basic world concepts
- There must be something better!

Pretraining

- To solve a new problem, you need new data
- Consider what this looks like: you spend most of your compute teaching the model basic world concepts
- There must be something better!
- Common practice is to use existing models and finetune them (E.g. VGGNet, which was then ludicrously large at 138M parameters). Still trained on supervised data! (ImageNet!)

Pretraining

- Can we, instead, train on unsupervised data, like, say, all of YouTube?
- This was done in 2012 at Google Brain!
- The authors used a sparse autoencoder trained on images, and found it could recognize various concepts
- If we can train our models on large corpuses of general data, we can then develop good representations, and finetune to specific tasks.

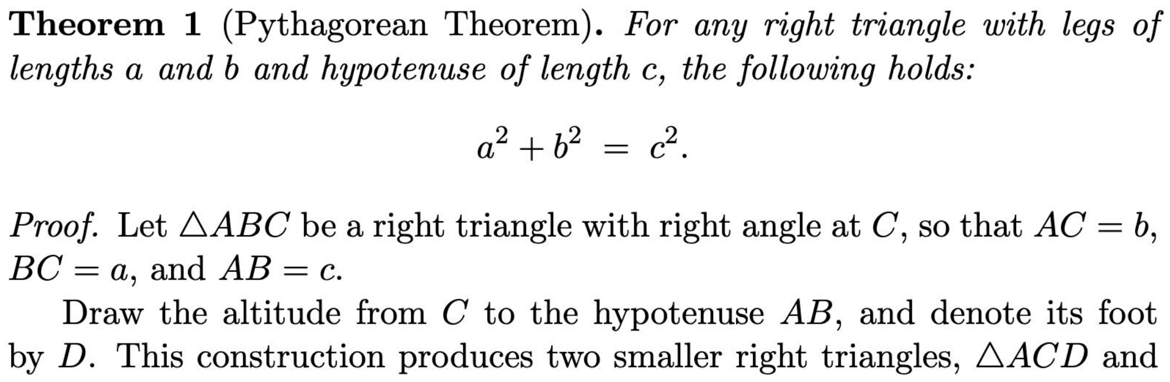
A history of pretraining

The basic idea:

- We have lots of compute
- We don’t have very much labeled data
- What can we do?

Next token prediction

- What is the minimum amount of information we need to make an accurate prediction?
- Consider the following examples:
    - “Mary had a little”
    - “E = M”



Theorem 1 (Pythagorean Theorem). *For any right triangle with legs of lengths a and b and hypotenuse of length c, the following holds:*

$$a^2 + b^2 = c^2.$$

Proof. Let $\triangle ABC$ be a right triangle with right angle at C, so that $AC = b$, $BC = a$, and $AB = c$.

Draw the altitude from C to the hypotenuse AB, and denote its foot by D. This construction produces two smaller right triangles, $\triangle ACD$ and

A history of pretraining

The problem now becomes:

- existing models saturate with more data (CNNs) or become too expensive (LSTMs).
- Existing models have issues with long sequences, either forgetting, or becoming extremely expensive.

The transformer

- Centered around modelling sequences
- Inputs are tensors with shape (B, S, D) (or (B, S) for LLMs specifically).
- It turns out that many useful problems have this structure!
    - Text data, naturally
    - Image pixels, using raster order (left to right, top to bottom)

The transformer

- The transformer is basically a large MLP
- This means that, at lower levels of compute, it does worse than more specialized architectures, but with more compute, does better
- E.g. Vision transformers (ViTs) vs CNNs, or Diffusion transformers (DiTs) vs UNets– only better at sufficient scale.
- But– with sufficient scale, it becomes *very* good.

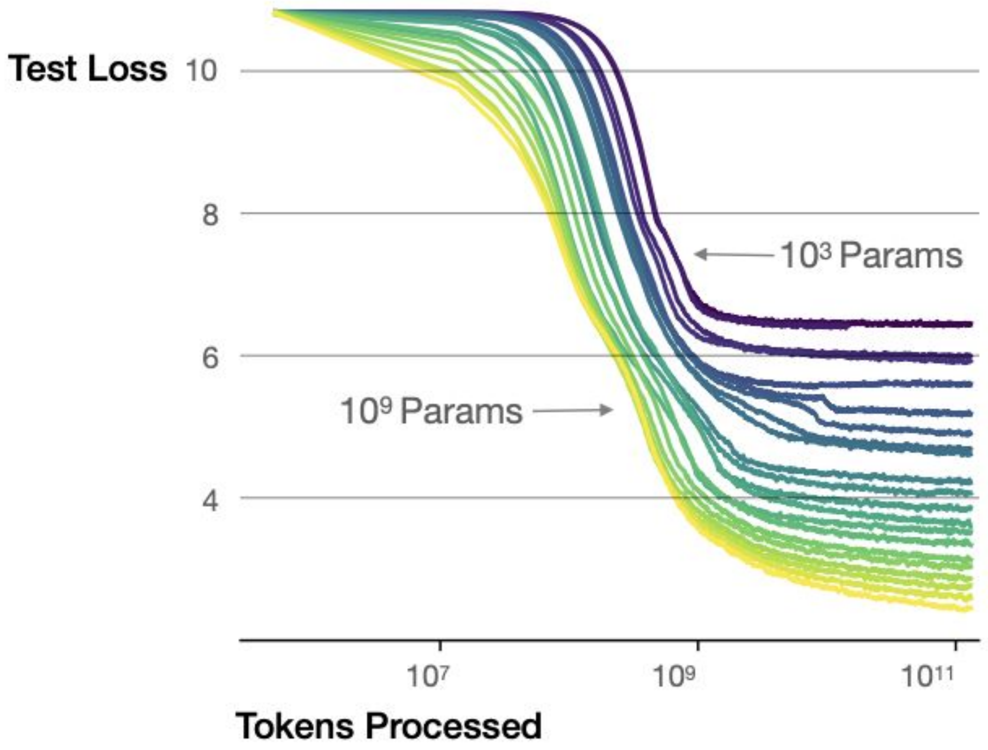
Scaling laws

Because scale was such a priority, this led to the development of *scaling laws*.

Two major papers:

1. Kaplan et. al, from OpenAI, and
2. Chinchilla, from DeepMind.

Larger models require **fewer samples** to reach the same performance

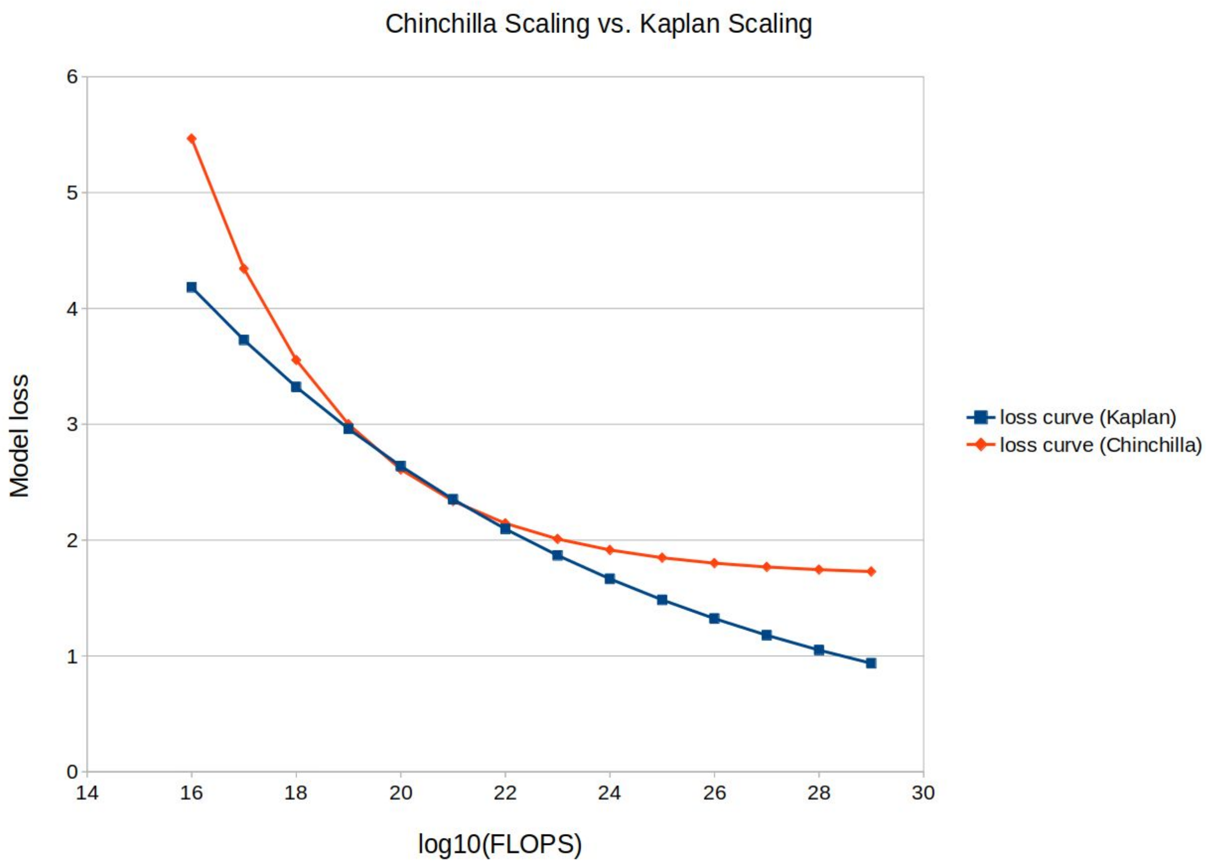


Kaplan vs Chinchilla

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

$$L(N,D) = \underbrace{\frac{A}{N^\alpha}}_{\text{finite model}} + \underbrace{\frac{B}{D^\beta}}_{\text{finite data}} + \underbrace{E}_{\text{irreducible}}$$

Two scaling laws, two conclusions



Kaplan’s conclusion

Model size is all you need.

Kaplan’s conclusion

Model size is all you need.

But…

This wasn’t entirely correct.

Chinchilla

Concluded:

- Data is the constraint in a lot of cases.
- Pushing model size above 300B parameters has very diminishing returns to scale

They trained a model which was better with “only” 70B params to show this.

Chinchilla

$$L(N,D) = \underbrace{\frac{A}{N^{\alpha}}}_{\text{finite model}} + \underbrace{\frac{B}{D^{\beta}}}_{\text{finite data}} + \underbrace{E}_{\text{irreducible}}$$

Chinchilla

$$L(N, D) = \underbrace{\frac{406.4}{N^{0.34}}}_{\text{finite model}} + \underbrace{\frac{410.7}{D^{0.28}}}_{\text{finite data}} + \underbrace{1.69}_{\text{irreducible}}$$

Chinchilla

If we insert the values for Gopher…

$$L(280 \cdot 10^9,\ 300 \cdot 10^9) = \underbrace{0.052}_{\text{finite model}} + \underbrace{0.251}_{\text{finite data}} + \underbrace{1.69}_{\text{irreducible}} = 1.993$$

The transformer

What *is* the transformer?

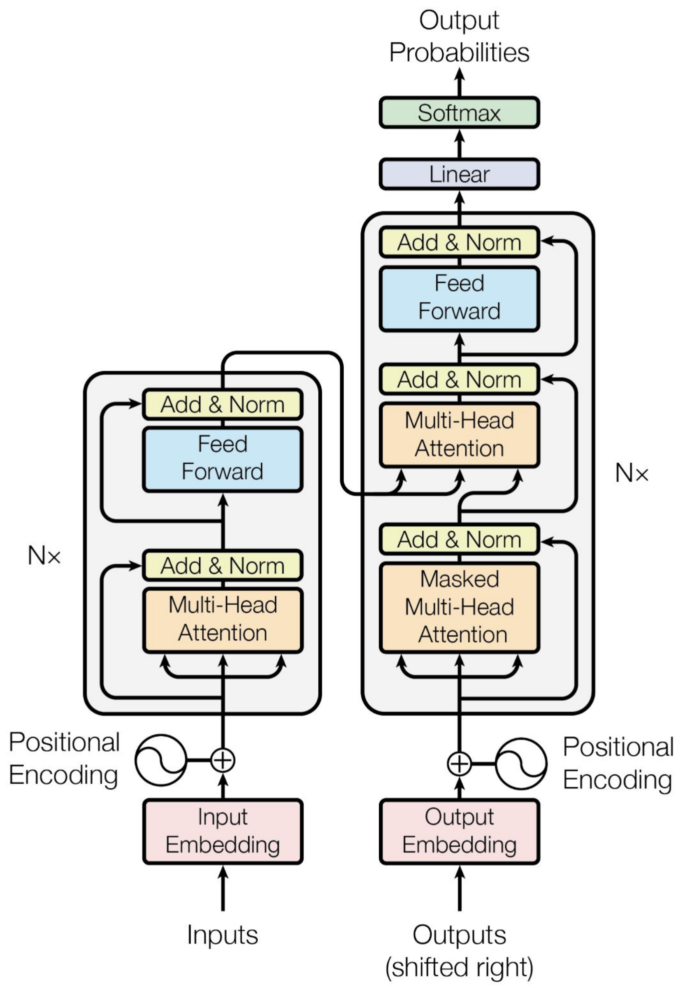


Figure 1: The Transformer - model architecture.

The transformer

Classic transformer has two parts:

1. The encoder, which attends over all tokens in the prompt
2. The decoder, which uses causal attention (each token only attends to the previous tokens).

As almost all modern transformers use a variant on the decoder, we focus on that.

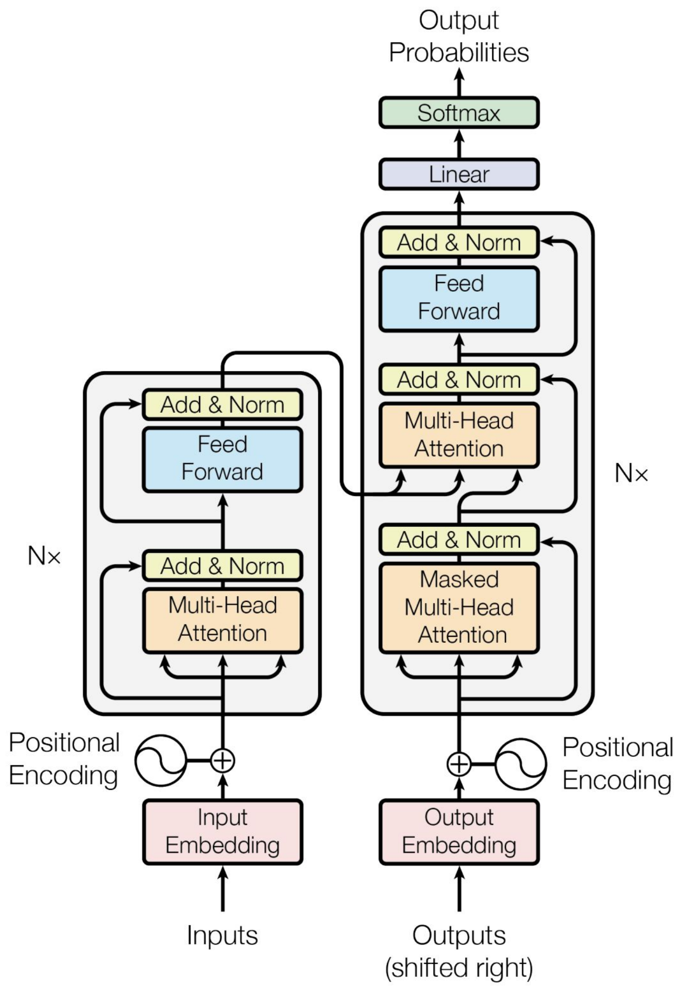
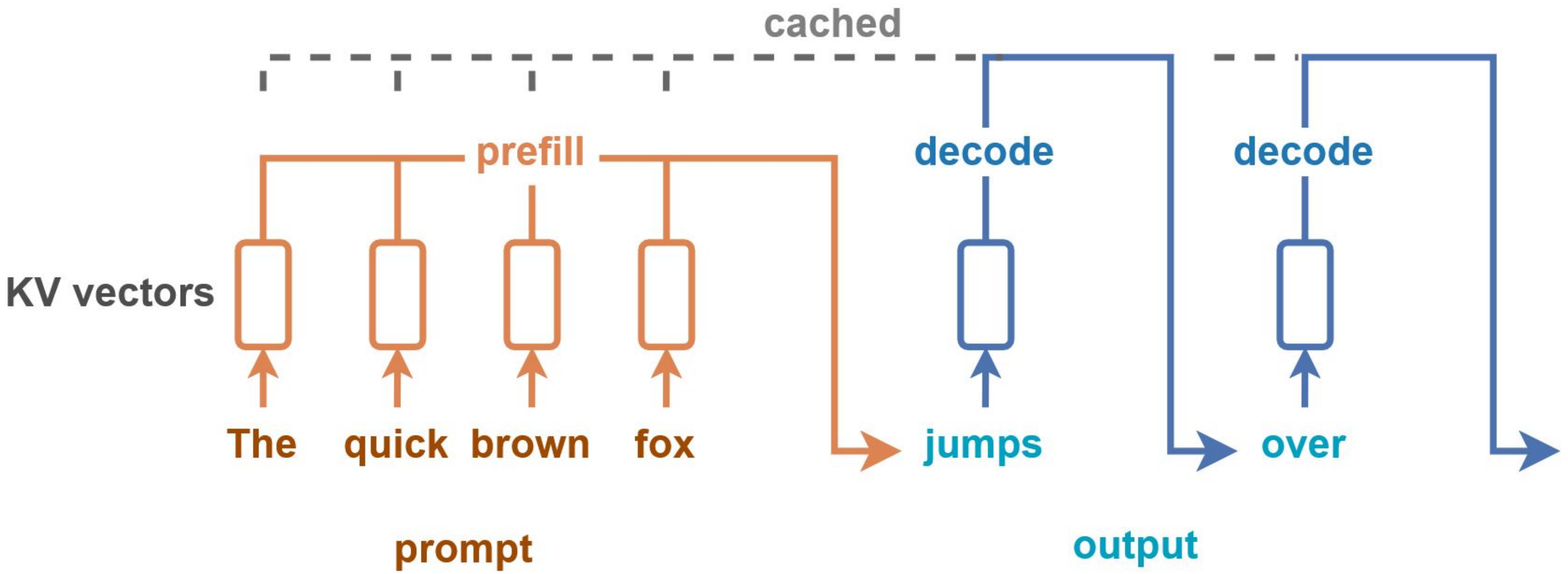


Figure 1: The Transformer - model architecture.

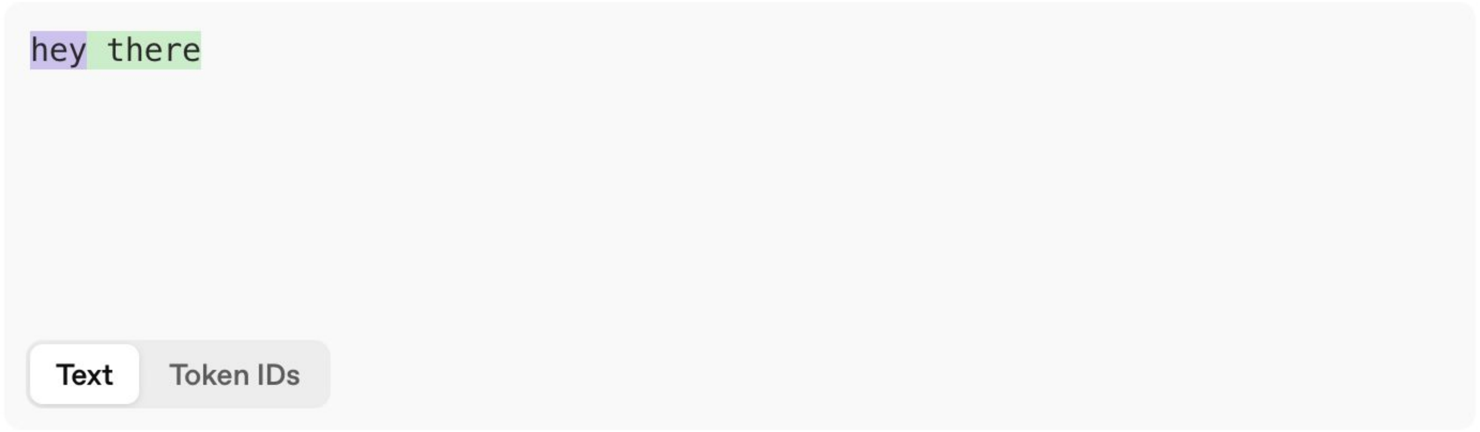

cached
prefill
decode
decode
KV vectors
The
quick
brown
fox
jumps
over
prompt
output

Tokenization

The inputs to transformers are (B, S) integer tensors of *tokens*.

Tokens are a numerical representation of text. Think: A=1, B=2, etc.

In practice: much more efficient!



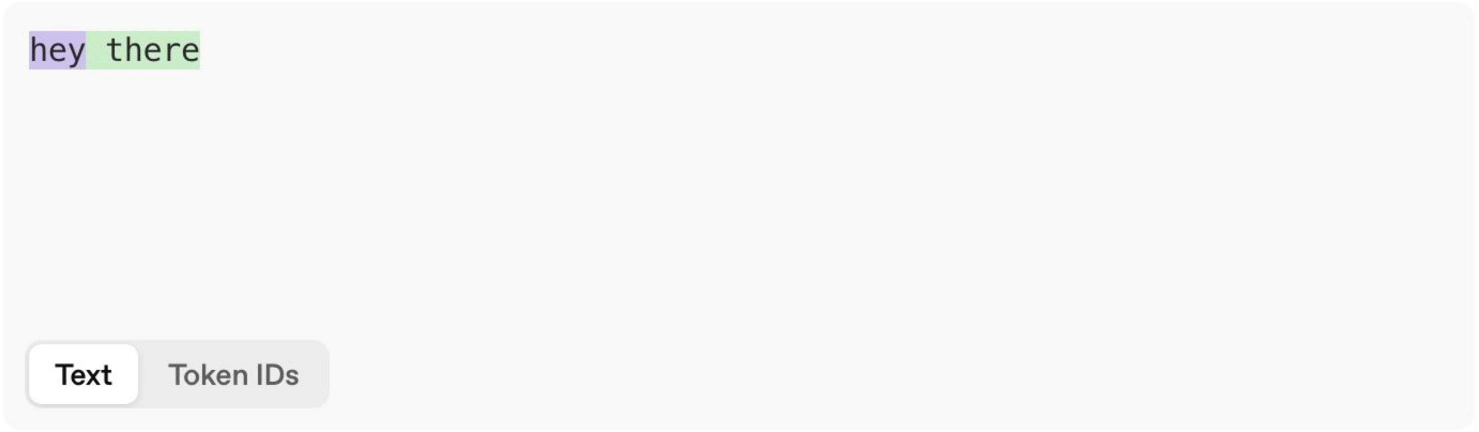
```
hey there
```

Tokenization

Motivation is that, by compressing the inputs, we can learn more.

Instead of “hey there” as [“h”, “e”, “y”, “ “, “t”, “h”, “e”, “r”, “e”],

we have [48467, 1354]. 4.5x more efficient! Can learn from 4.5x the data with the same compute.


hey there

Tokenization

This *can* be done for non-text modalities, like images or audio, but more typically, pass raw pixels in. For images (passed to ViTs), we do the following:

$$
\begin{aligned}
(H, W, C) &\rightarrow \left(\frac{H}{P_h}\frac{W}{P_w}, P_h, P_w, C\right) \\
&\rightarrow (N, P_h \cdot P_w \cdot C) \\
&= (S, L)
\end{aligned}
$$

And typically videos are represented as sequences of images.

The decoder (transformer)

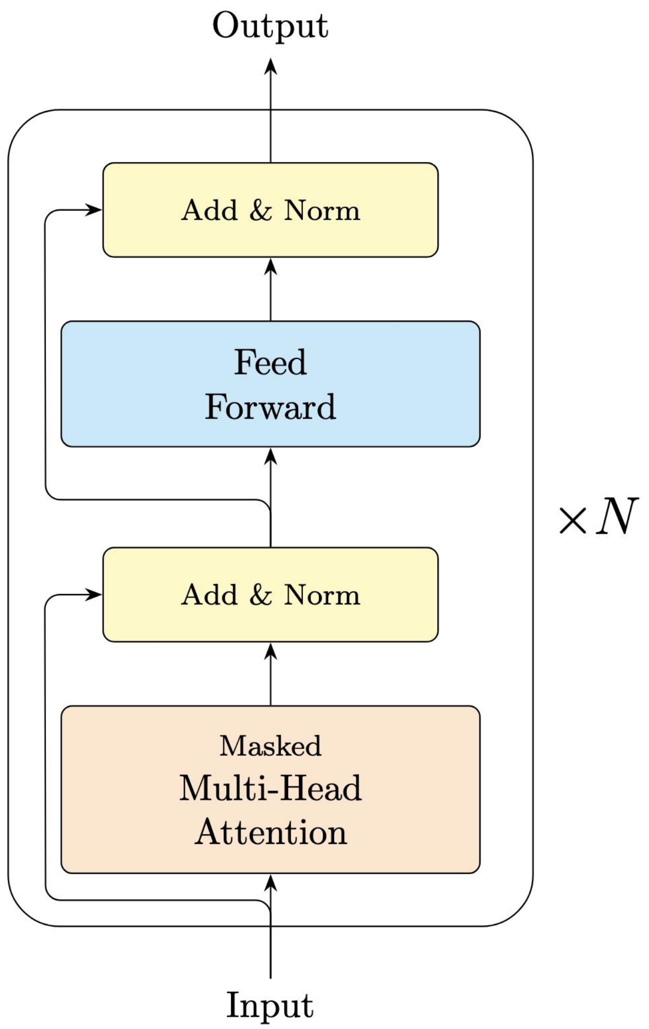
The decoder LLM is a deep neural network consisting of:

1. An embedding layer, which converts the sequence of integers into embeddings. This is a (vocabulary_size, embedding_dim) matrix.
2. N successive decoder blocks, which take in and output
3. An output head mapping the final block activations into a probability distribution over the vocabulary. This is a (embedding_dim, vocabulary_size) matrix.
   a. In many implementations, the output head and the embedding layer are identical. These matrices can be very large– GPT3, for instance, had a 12288 embedding_dim and a vocabulary size of 50,257. Using 32bit floating point numbers, that’s 2.5GB per matrix (600M parameters).

The decoder block

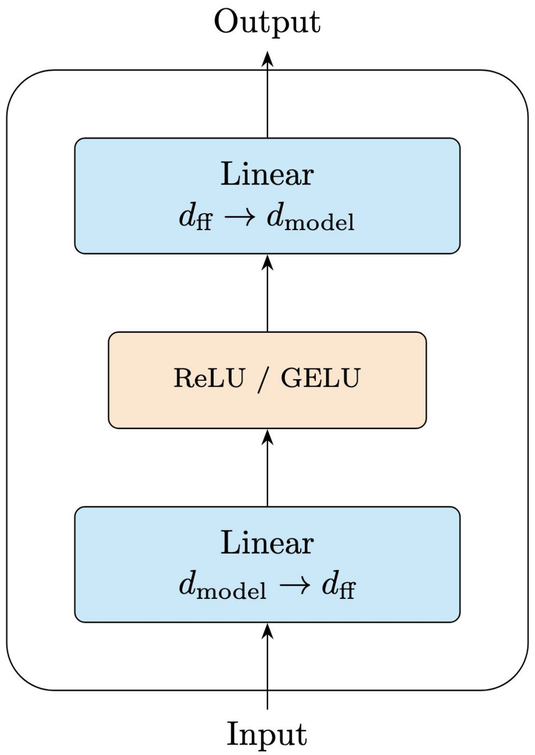
The decoder block is straightforward:

1. Take the (B, S, D) embedding as input
2. Run through an attention layer followed by a residual connection
3. Run through a feed forward layer followed by a residual
4. Depending on the *specific* architecture, apply normalizations.



Feed forward

Standard 2 layer MLP!


Output
Linear
$d_{\text{ff}} \rightarrow d_{\text{model}}$
ReLU / GELU
Linear
$d_{\text{model}} \rightarrow d_{\text{ff}}$
Input

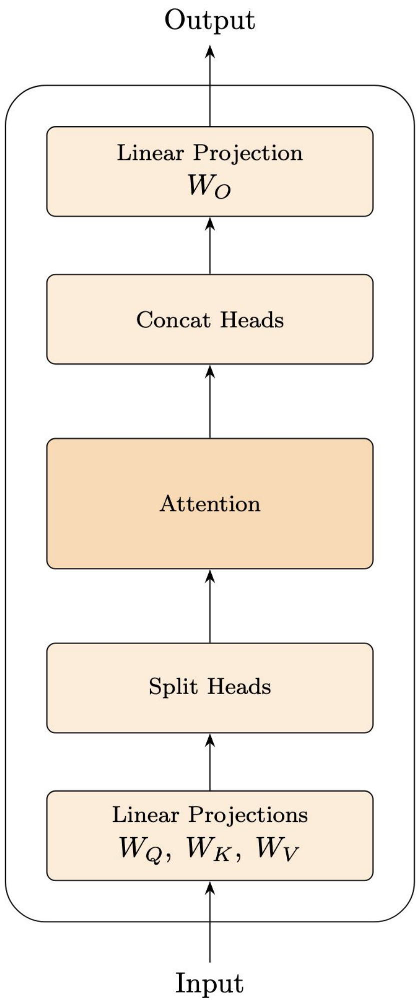
Multi-head attention?

Inputs are (B, S, D) tensors.

W_{q, k, v} are (D, D) matrices, so Q, K, V are (B, S, D) tensors.

We split these into (B, H, S, D//H) tensors.

Do B * H attention calculations on the (S, D // H) tensors, combine the results.



Attention

$$\mathbf{Attention}(Q, K, V) = \mathbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Attention

- Develop a *sequence-wise* representation, with dependencies on all other elements in the sequence
- `softmax(QK^T)` *weights* V
- Q, K, and V are all the same– namely, embeddings of the previous layer activations– we are mixing the representations and allowing for interactions.

$$\mathbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Attention

$$Q, K, V \in R^{B \times H \times S \times D_h}$$

$$\text{(1)Dot-product: } QK^\top : (B, H, S, D_h) \times (B, H, D_h, S) \longrightarrow (B, H, S, S)$$

$$\text{Cost } = B\,H\,S\,D_h\,S = B\,H\,D_h\,S^2$$

$$\text{(2)Weighting values: } \text{Attn}(QK^\top)\,V : (B, H, S, S) \times (B, H, S, D_h) \longrightarrow (B, H, S, D_h)$$

$$\text{Cost } = B\,H\,S\,S\,D_h = B\,H\,D_h\,S^2$$

$$\text{Total cost} = 2\,B\,H\,D_h\,S^2$$

$$\text{With } D = H \cdot D_h \text{ (model embedding dim)}$$

$$\boxed{\mathcal{O}\big(B\,D\,S^2\big)}$$

Attention variants

$$\begin{aligned}\mathbf{Attention}(Q, K, V) &= \mathbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \\ &= \mathbf{weight}(Q, K)V\end{aligned}$$

Attention variants

$$\mathbf{Attention}(Q, K, V) = \mathbf{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

$$= \mathbf{weight}(Q, K)V$$

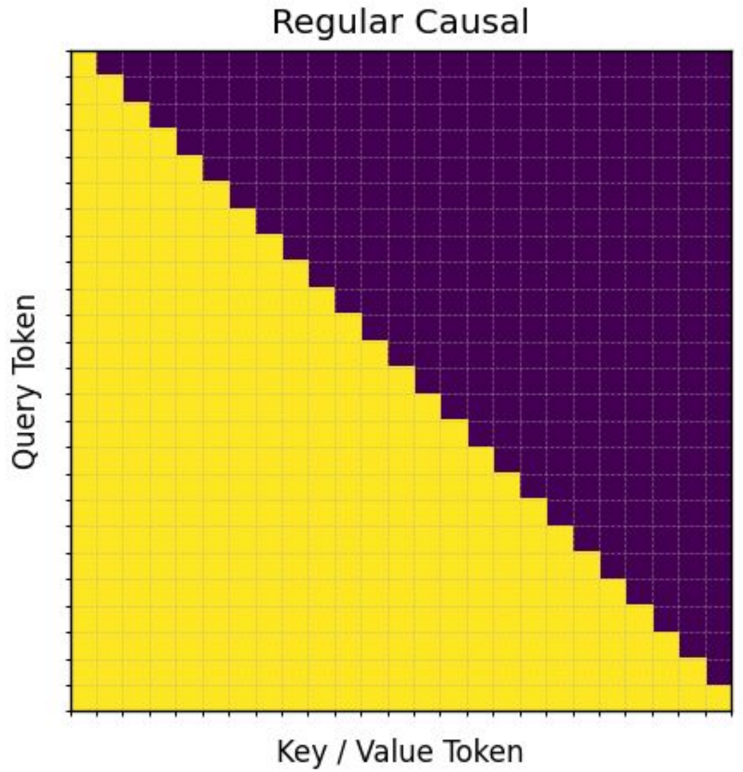
If we use alternate weighting functions, can get rid of quadratic complexity!

Standard practice is to have M sparse attention layers followed by 1 global attention layer.

Standard (full) attention

This works really well, but is expensive.

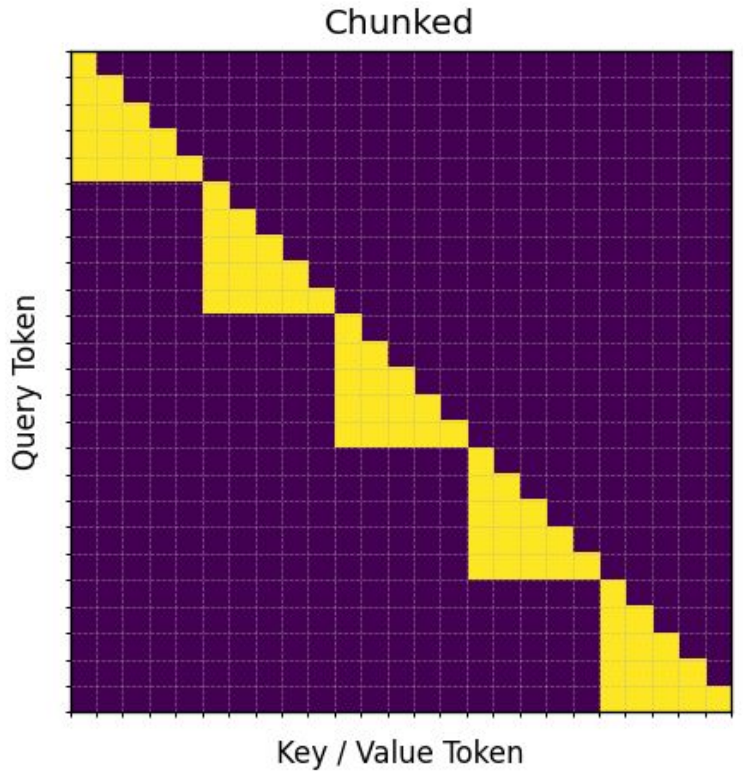
Quadratic in sequence length!



Chunked attention

Very computationally efficient, but doesn’t work well at the boundaries.

Thought to be partially responsible for Llama4’s issues.

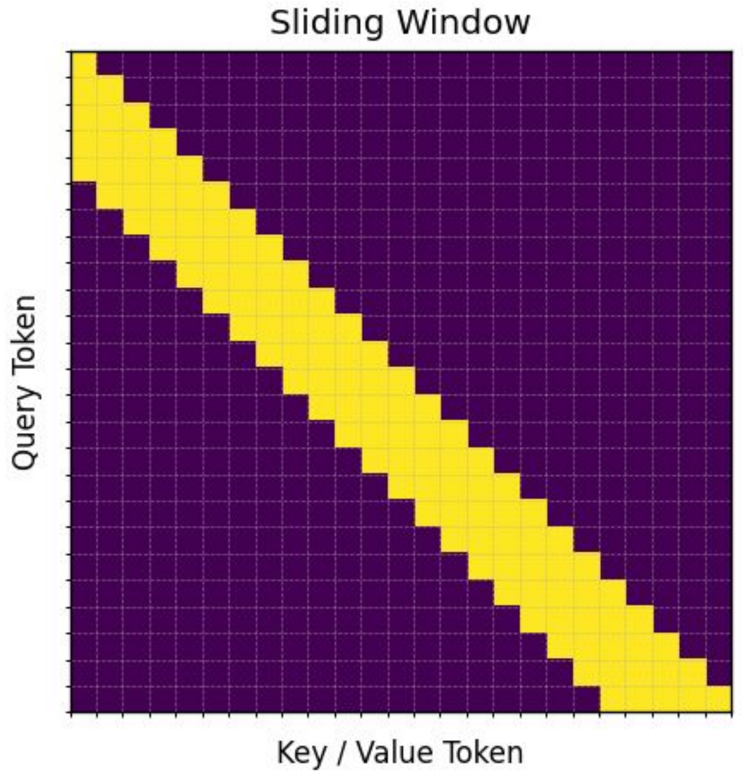


Sliding window attention

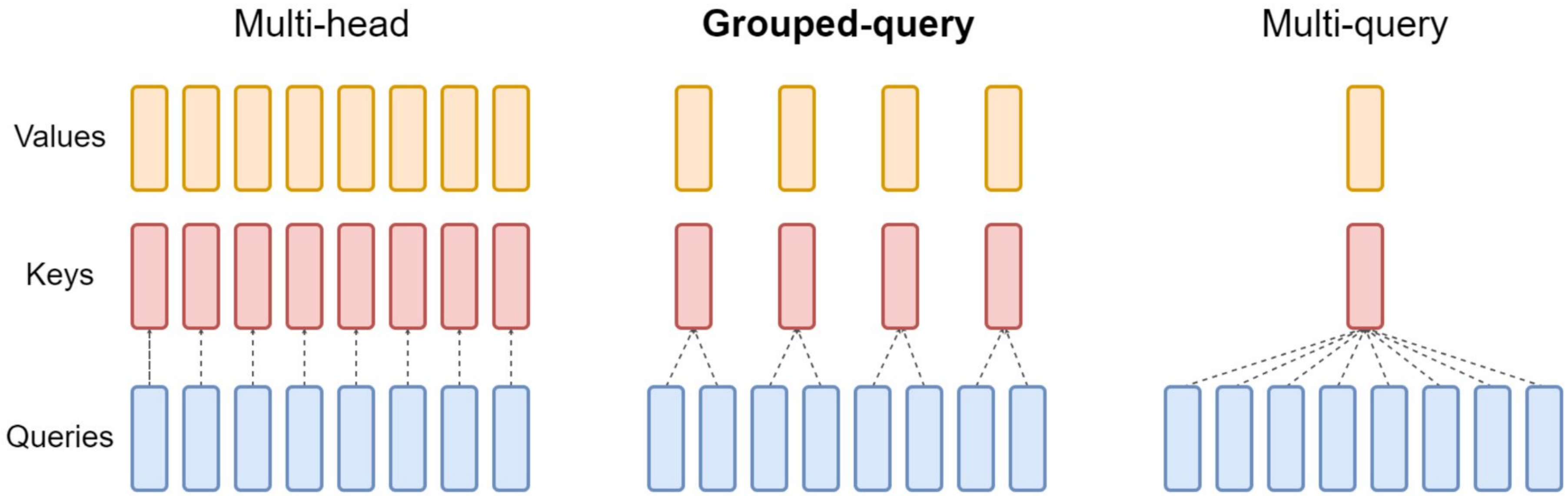
This works fairly well, but is annoying to compute.

A standard inference optimization is to cache the K, V values (often the most expensive part of attention calculation!).

Non-trivial to do that here.



Do we actually need multiple heads?



Normalization

Normalization varies; 2 main flavors:

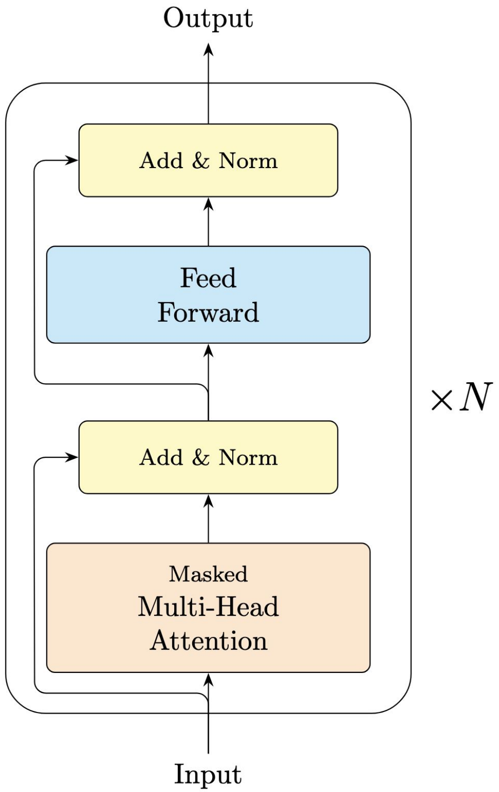
1. Pre-norm, in which we apply the normalization *before* the sublayers:

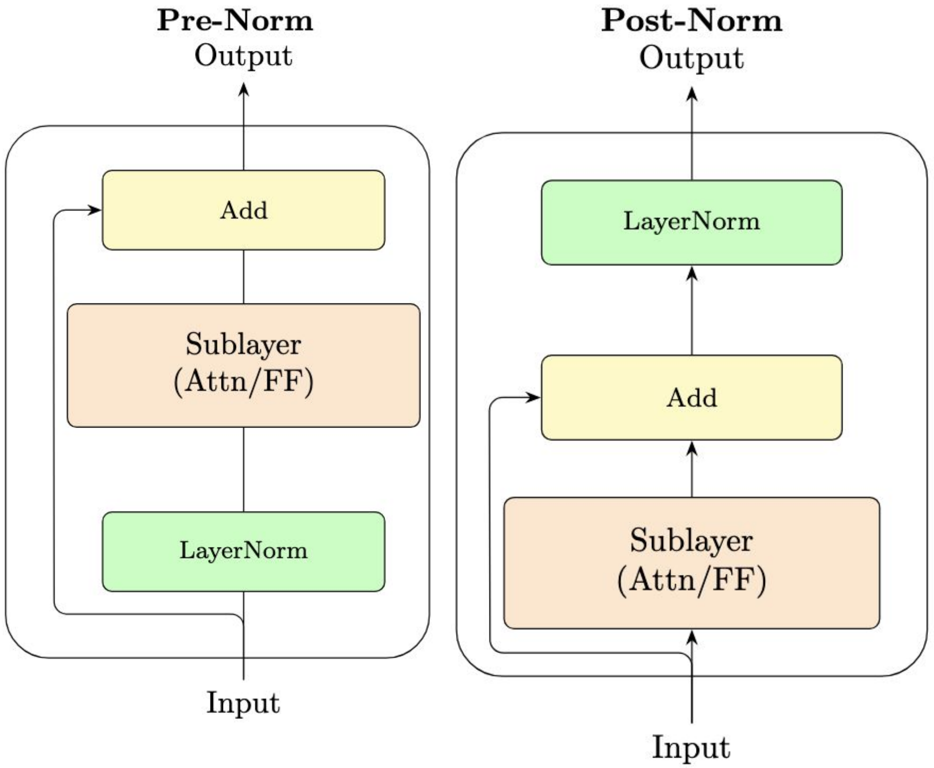
```
# Pre-LN
def forward(self, x):
    x = x + self.attn(self.attn_ln(x))   # LN before sub-layer
    x = x + self.ff(self.ff_ln(x))       # LN before MLP
    return x                             # (often add a final LN after the last block)
```

2. Post-norm, in which we apply it *after* the sublayers:

```
# Post-LN
def forward(self, x):
    x = self.attn_ln(x + self.attn(x))   # LN after residual add
    x = self.ff_ln(x + self.ff(x))       # LN after residual add
    return x
```

Normalization



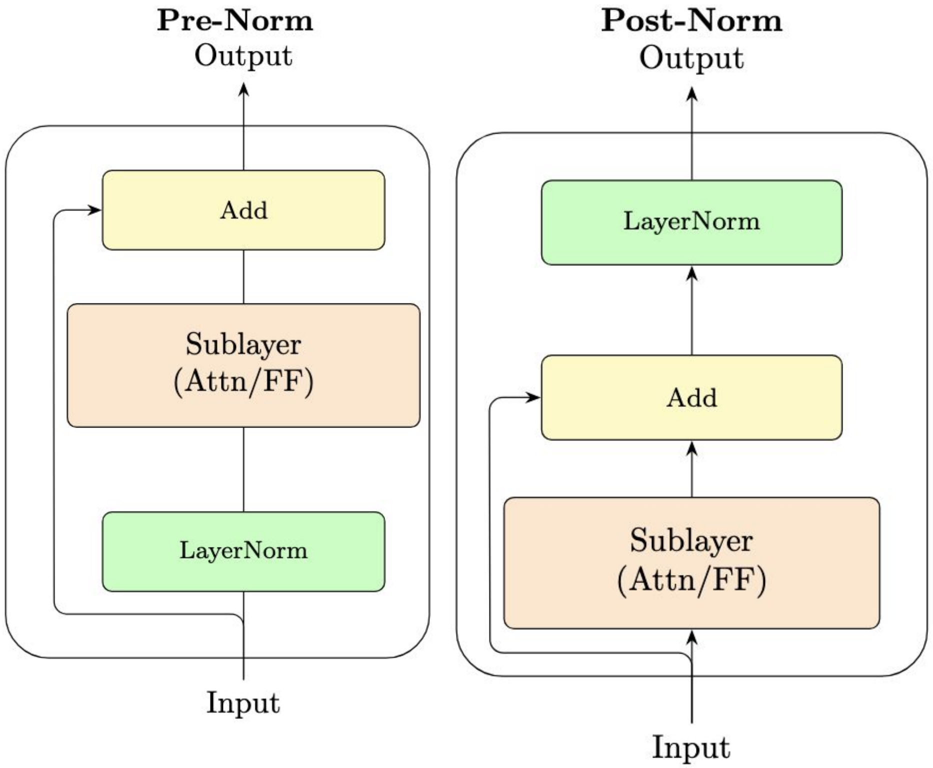


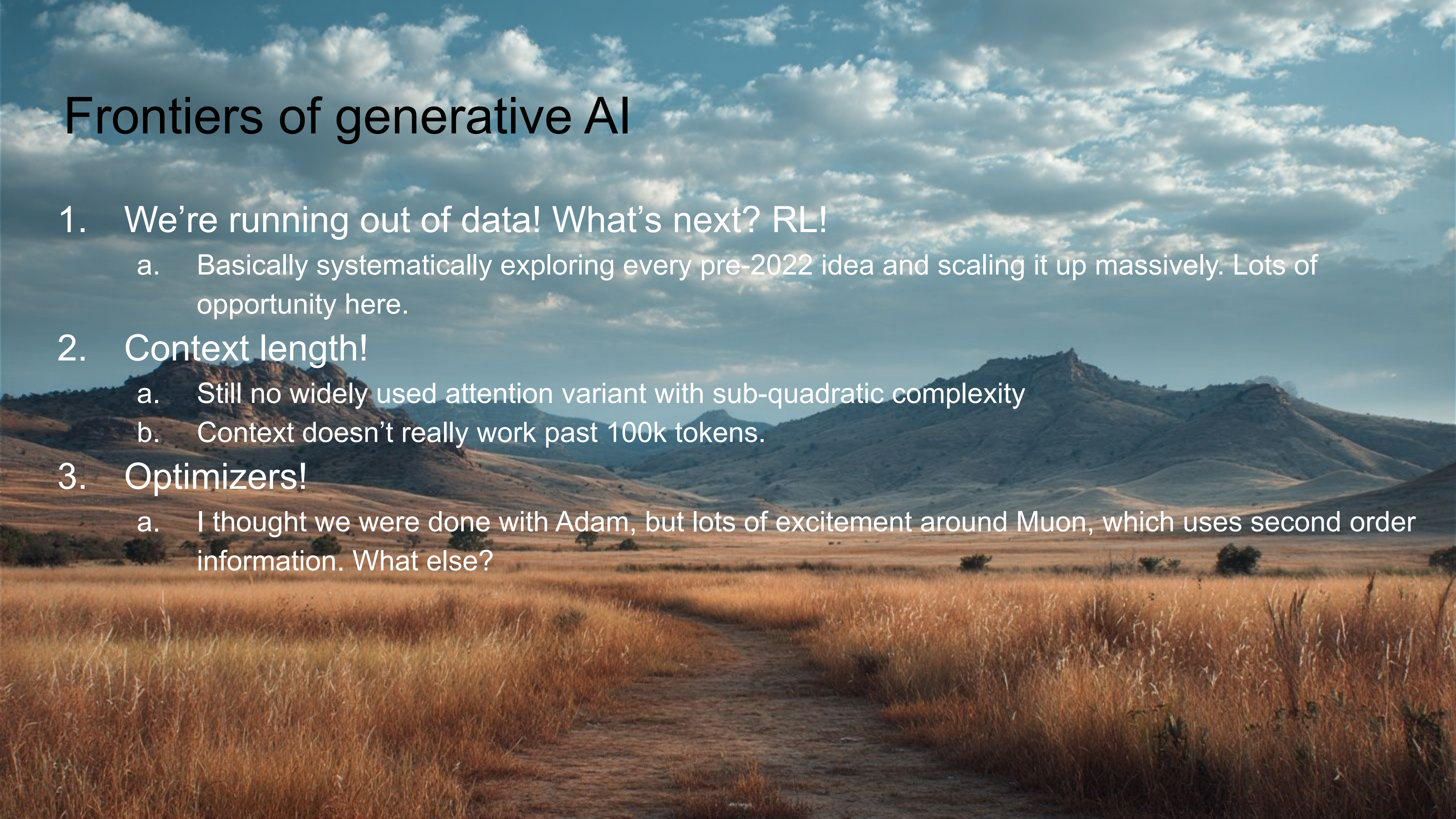
Normalization

Why does this matter?

Consider the path the gradient takes:

1. With pre-norm, there's a straight shot from the end to the beginning
2. With post-norm, by the time the gradient reaches the initial (embedding) layer, has gone through Nx LayerNorms. So either shrinks/explodes!





Frontiers of generative AI

1. We're running out of data! What's next? RL!
   a. Basically systematically exploring every pre-2022 idea and scaling it up massively. Lots of opportunity here.
2. Context length!
   a. Still no widely used attention variant with sub-quadratic complexity
   b. Context doesn't really work past 100k tokens.
3. Optimizers!
   a. I thought we were done with Adam, but lots of excitement around Muon, which uses second order information. What else?

RL with LLMs

Two major flavors:

1. RL on human feedback (RLHF), in which we gather human data, typically pairs of samples, train a reward model, and optimize that.
2. RL with verifiable rewards, in which the model generates a bunch of data, which is then verified, and a reward is assigned. Standard RL!

Extremely under-explored! Basic RL ideas have yet to be explored. Very little work on e.g. replay buffers.

Questions?

✉ finbarrtimbers@gmail.com

𝕏 @finbarrtimbers

🌐 https://finbarr.ca/transformers