TRANSFORMER FLOPS

Adam Casson
me@adamcasson.com

May 16, 2023
It is recommended to read this in the original HTML format:
https://www.adamcasson.com/posts/transformer-flops

ABSTRACT

Counting the number of floating-point operations (FLOPs) in Transformers is a useful way to estimate compute requirements and measure efficiency. As training runs get larger and larger (thus more expensive) it becomes more important to understand how many FLOPs we need to do and how well we utilize our hardware.

1 Counting FLOPs in Transformers

One commonly used method for counting FLOPs is from the OpenAI scaling law paper [Kaplan et al., 2020] which uses

$$C_{\text{forward+backward}} \approx 6N$$

for estimating the number of FLOPs per token during the training of a decoder-only Transformer where N is the number of non-embedding parameters in the model. To derive this we can look at Table 1 they provide for FLOP counts of various components of the model for the forward pass.

Nonlinearities, biases, normalizations, and residuals are not counted as they turn out to be negligible. Let's explain each operation and variable here:

- `Embed`: learned token embeddings and learned positional embeddings.
  - d_{model} is the dimensionality of the residual stream.
- `Attention: QKV`: linear layer in multi-head self-attention to project input into queries, keys, and values.
  - n_{layer} is the number of layers.
  - d_{attn} is the dimensionality of the output of multi-headed attention, which is equal to $d_{\text{key}}\, n_{\text{heads}}$.
    - d_{key} is the dimension of the key, query, and value projections.
    - n_{heads} is the number of attention heads in a layer.
    - In practice, Transformers are implemented such that $d_{\text{attn}} = d_{\text{model}}$.
- `Attention: Mask`: dot-product between query and keys.
  - n_{ctx} is the context/sequence length.
- `Attention: Project`: linear layer to project concatenated attention heads output to d_{model}.
- `Feedforward`: two linear layers in the MLP block.
  - d_{ff} is the size of the output dimensionality of the first linear layer.
    - $d_{\text{ff}} = 4\, d_{\text{model}}$ is commonly used.
- `De-embed`: linear layer to obtain logits over vocabulary.
  - n_{vocab} is the number of tokens in the vocabulary.

Table 1: Kaplan et al. [2020] FLOPs per token.

Operation	**Parameters**	**FLOPs per Token**
Embed	$\left(n_{\text{vocab}} + n_{\text{ctx}}\right) d_{\text{model}}$	$4\, d_{\text{model}}$
Attention: QKV	$n_{\text{layer}}\, d_{\text{model}}\, 3\, d_{\text{attn}}$	$2\, n_{\text{layer}}\, d_{\text{model}}\, 3\, d_{\text{attn}}$
Attention: Mask	$-$	$2\, n_{\text{layer}}\, n_{\text{ctx}}\, d_{\text{attn}}$
Attention: Project	$n_{\text{layer}}\, d_{\text{attn}}\, d_{\text{model}}$	$2\, n_{\text{layer}}\, d_{\text{attn}}\, d_{\text{model}}$
Feedforward	$n_{\text{layer}}\, 2\, d_{\text{model}}\, d_{\text{ff}}$	$2\, n_{\text{layer}}\, 2\, d_{\text{model}}\, d_{\text{ff}}$
De-embed	$-$	$2\, d_{\text{model}}\, n_{\text{vocab}}$
Total (Non-Embedding)	$N \;=\; 2\, d_{\text{model}}\, n_{\text{layer}}\left(2\, d_{\text{attn}} + d_{\text{ff}}\right)$	$C_{\text{forward}} \;=\; 2\, N \;+\; 2\, n_{\text{layer}}\, n_{\text{ctx}}\, d_{\text{attn}}$

Combining all the non-embedding FLOPs terms gets us

$$\begin{aligned} C_{\text{forward}} &= 2\, n_{\text{layer}}\, d_{\text{model}}\, 3\, d_{\text{attn}} \;+\; 2\, n_{\text{layer}}\, d_{\text{attn}}\, d_{\text{model}} \\ &\quad +\; 2\, n_{\text{layer}}\, 2\, d_{\text{model}}\, d_{\text{ff}} \;+\; 2\, n_{\text{layer}}\, n_{\text{ctx}}\, d_{\text{attn}} \\ &= 2\big(2\, d_{\text{model}}\, n_{\text{layer}}\, (2\, d_{\text{attn}} + d_{\text{ff}})\big) \;+\; 2\, n_{\text{layer}}\, n_{\text{ctx}}\, d_{\text{attn}} \\ &= 2\, N \;+\; 2\, n_{\text{layer}}\, n_{\text{ctx}}\, d_{\text{attn}}. \end{aligned}$$

for the forward pass. They note though that they drop the last term, which is context-dependent, because when $d_{\text{model}} > n_{\text{ctx}}/12$ it becomes negligible.

We can see this with a quick calculation using GPT-3 ($n_{\text{layer}} = 96$, $n_{\text{ctx}} = 4096$, $d_{\text{attn}} = 12288$) as an example

$$\begin{aligned} C_{\text{forward}} &= 2\big(175 \times 10^9\big) \;+\; 2 \cdot 96 \cdot 4096 \cdot 12288 \\ &= \big(350 \times 10^9\big) \;+\; \big(9.7 \times 10^9\big). \end{aligned}$$

which shows that the context-dependent term makes up $< 3\%$ of the FLOP count. So the number of FLOPs can be simplified to just

$$C_{\text{forward}} \approx 2N$$

The factor of 2 can be explained by the fact that matmuls consist of a 2 FLOP multiply-accumulate operation (one multiply and one add) for each element in a weight matrix. We can then realize that that the backward pass must account for another $4N$ of FLOPs since we need to do twice the matmuls that we do during the forward pass [Bahdanau, 2022]. This gets us to the $C_{\text{forward+backward}} \approx 6N$ equation.

We can then multiply by some number of tokens D to estimate the total FLOPs needed for training on those D tokens. Doing this we get $C = 6DN$.

Another method of calculating Transformer FLOPs is presented in DeepMind's Chinchilla scaling law paper [Hoffmann et al., 2022]. Table 2 shows their equations for forward pass FLOPs.

Like the first method, DeepMind also assumes the backwards pass has 2 times the FLOPs of the forward. Unlike OpenAI, DeepMind includes the FLOPs from embeddings and logits (de-embed) as well as the softmax in attention and the application of the attention pattern to the values. Also it's important to note that OpenAI's method is FLOPs *per token* while DeepMind's is FLOPs *per sequence*. This doesn't fundamentally change anything but it's important to remember in order to use either method correctly. Overall the difference between this method and $C = 6N$ tends to be pretty minimal as shown in Table A4 (Appendix F) of the Chinchilla paper.

If you want to count the FLOPs of your own transformer I've provided a FLOPs calculator app in Appendix A.

Table 2: Hoffmann et al. [2022] FLOPs per sequence.

Operation	**FLOPs per Sequence**
Embeddings	$2\, n_{\text{ctx}}\, n_{\text{vocab}}\, d_{\text{model}}$
Attention: QKV	$2\, n_{\text{ctx}}\, 3\, d_{\text{model}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: QK logits	$2\, n_{\text{ctx}}\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: Softmax	$3\, n_{\text{heads}}\, n_{\text{ctx}}\, n_{\text{ctx}}$
Attention: Reduction	$2\, n_{\text{ctx}}\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: Project	$2\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right) d_{\text{model}}$
Feedforward	$4\, n_{\text{ctx}} \left(d_{\text{model}}\, d_{\text{ff}}\right)$
Logits	$2\, n_{\text{ctx}}\, d_{\text{model}}\, n_{\text{vocab}}$
Total	Embeddings $+\ n_{\text{layers}} \times$ (Attention $+$ Feedforward) $+$ Logits

2 Using FLOPs to measure efficiency

Counting the number of FLOPs in our models grounds us in the reality of how much raw compute we need to run them. As training runs and models get bigger and bigger it means, of course, that training and serving these models gets more expensive. As we occupy precious resources like a A100s or H100s it becomes important to try to crunch the numbers that we *need* to crunch as quickly as the hardware will physically allow (emphasis on the "need" which we'll get to). To get a sense of this we will want to look at how many FLOPs/second (FLOPS, with a big S, will be used to refer to floating point operations *per second* and FLOPs, with a little s, is used for floating point operations with no unit of time) we execute and compare it against the theoretical peak FLOPS of our hardware. In practice, we'll never be able to achieve that peak, but it's useful to know how far away we are from it.

One method that can be used for measuring training efficiency is hardware FLOPS utilization (HFU). This approach is the ratio of all FLOPs we execute per second to the theoretical peak FLOPS. This would take into account all the FLOPs we estimate for a regular forward+backward pass, but also redundant computation we need to do in order to train large models on the current hardware like rematerialization for activation checkpointing [1]. While this method of measurement can be useful, the inclusion of computation like rematerilization can make it seem like our training is more efficient than it really is. Ultimately, if we could train these models without tricks like activation checkpointing we would. It's only there as a work around due to constraints of current hardware and being able to eliminate it one day would be efficient. What we really care about is only the FLOPs we *need* to do to train the model in theory which is just the forward+backward FLOPs, or the *model* FLOPs.

The best practice now for reporting LLM training efficiency is known as model FLOPs utilization (MFU) which was proposed in Google's PaLM paper [Chowdhery et al., 2022]. The idea is to focus on how efficiently we executed just the necessary *model* FLOPs. The calculation is quite simple as all we need to do is multiply our FLOPs count by the observed throughput (tokens/sequences per second), and then divide by the theoretical peak FLOPS of our hardware.

$$MFU = \frac{CD}{P}$$

where C is the model's FLOPs per token, D is the observed tokens per second, and P is the theoretical peak FLOPS.

For example, when using the fp16/bf16 formats an A100 has a theoretical peak of 312 teraFLOPS (TFLOPS)[2]. If we use the $6N$ estimate for forward+backward FLOPs and are training a 125M parameter model using an A100 and we have throughput of 200,000 tokens per second then our MFU is

[1] If we can't store activations in memory then we can do activation checkpointing by throwing out the activations after they aren't needed anymore in the forward pass and then re-computing (a.k.a *rematerializing*) them during the backward pass. If done for all activations this would amount to an extra forward pass and thus our $C = 6N$ would increase to $C = 8N$.

[2] NVIDIA A100 Datasheet

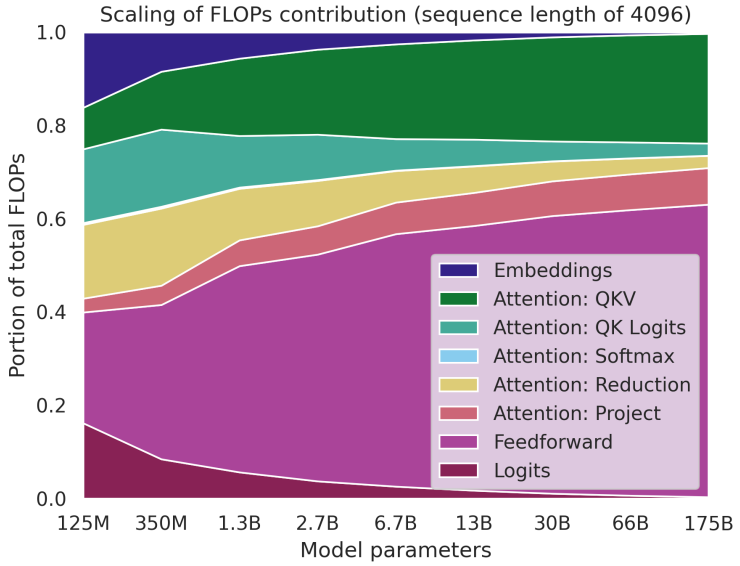
$$\begin{aligned}\text{MFU} &= \frac{6ND}{P} \\ &= \frac{6 \cdot 125 \times 10^6 \cdot 200 \times 10^3}{312 \times 10^{12}} \\ &= 0.48.\end{aligned}$$

Which means we achieved 48% of the theoretical peak FLOPS. With this method if we are doing something like activation checkpointing this will hurt our MFU, but it wouldn't necessarily hurt our HFU. Conversely, if we could get away with not using activation checkpointing then our MFU would improve. This isn't to say HFU is worthless though. If we *need* to use activation checkpointing, HFU can help gauge the efficiency of our rematerilization implementation. In the long run though we should generally want to optimize for MFU and it also allows fairer comparison of efficiency across different training set ups.

In practice, the range of MFU for language models can vary widely depending on model size, hardware[3], and implementation details, but generally range between $10 - 65\%$ [Portes et al., 2023][4][5][6].

3 Scaling of FLOPs

As we scale the size of Transformers it can be useful to know how different components of the model contribute to the computational cost [Roller, 2022, He, 2022, Timbers, 2023]. Let's look at how the FLOPs of these components (using the operations in the DeepMind table) contribute to the total compute as we scale the model.



The plot above shows this evolution using the GPT-3[Brown et al., 2020]/OPT[Zhang et al., 2022] model family (with a sequence length of 4096). As we can see, the `Embeddings` and `Logits` become a very miniscule portion of FLOPs while the matmuls of the linear layers in `Attention: QKV` and `Feedforward` become dominant.

A useful way to look at this is to divide the components of the model into two buckets: terms that scale linearly with sequence length and those that scale quadratically. The terms that scale quadratically with sequence length are `Attention: QK logits`, `Attention: Softmax`, and `Attention: Reduction`. All other terms (`Embeddings`, `Attention: QKV`, `Attention: Project`, `Feedforward` and `Logits`) scale linearly with sequence length.

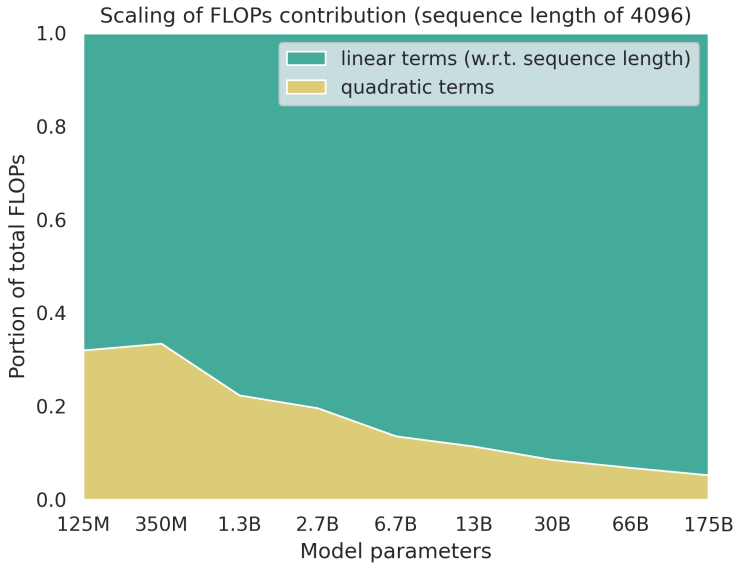
[3] Although MFU normalizes according to a hardware's theoretical peak FLOPs there can be many factors that impact this including memory-bandwidth, cross-device/cross-node communication bandwidth, etc.

[4] See section *Multinode Scaling of MosaicBERT*.

[5] MosaicGPT Training Benchmarks

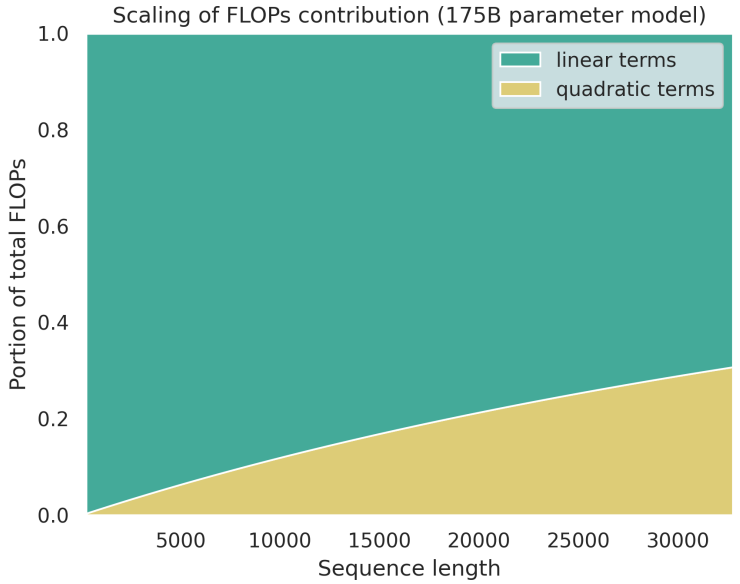
[6] See section 4.1 of PaLM [Chowdhery et al., 2022]

We can calculate FLOPs for various model sizes and look at how these terms evolve as the number of parameters increases.



For the smallest models the quadratic attention terms make up over 30% of the FLOPs, but this steadily decreases. Somewhere between 13B and 30B it starts to account for $< 10\%$ of FLOPs. The dominance of the "linear" terms is explained by the fact that the size of the weight matrices in linear layers *does* scale quadratically but with respect to d_{model}. So by making our model *wider* we know that we'll be increasing the portion of FLOPs that come from the linear layers.

We can also look at the scaling of these components from the view of a fixed model size but varying sequence length.



This shows for a 175B parameter model how much the quadratic terms contribute to total FLOPs as sequence length scales. For the smallest sequence lengths (i.e. 256 or 512) the quadratic terms make up $< 1\%$ of FLOPs. After the 8192 mark they then make up $> 10\%$ and then hit 31% for a 32K length sequence.

So at large scales the quadratic nature of attention only makes up a fraction of the model's FLOPs even at longer sequence lengths.

Although the scope of this post is FLOPs bound (pun intended), when it comes to profiling and optimizing models there is much more to consider. Here are some helpful resources to read more in that direction:

- Transformer Inference Arithmetic
- Making Deep Learning Go Brrrr From First Principles
- Data Movement Is All You Need: A Case Study on Optimizing Transformers

References

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Dzmitry Bahdanau. The FLOPs calculus of language model training. https://medium.com/@dzmitrybahdanau/the-flops-calculus-of-language-model-training-3b19c1f025e4, 2022. Medium blog post. Accessed: May 2023.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. URL https://arxiv.org/abs/2204.02311.

Jacob Portes, Alex Trott, Daniel King, Sam Havens, and Erica Ji Yuen. Mosaicbert: Pretraining bert from scratch for $20. https://www.mosaicml.com/blog/mosaicbert, 2023. MosaicML Blog. Accessed: May 2023.

Stephen Roller. Twitter post. https://twitter.com/stephenroller/status/1579993017234382849, 2022. URL https://twitter.com/stephenroller/status/1579993017234382849.

Horace He. Twitter post. https://twitter.com/cHHillee/status/1579913387021979649, 2022. URL https://twitter.com/cHHillee/status/1579913387021979649.

Finbarr Timbers. Twitter post. https://twitter.com/finbarrtimbers/status/1643621208112390147, 2023. URL https://twitter.com/finbarrtimbers/status/1643621208112390147.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022. URL https://arxiv.org/abs/2205.01068.

A Transformer FLOPs calculator

View the source code for this app here.

https://huggingface.co/spaces/adamcasson/transformer-flops-calculator

B FLOPs counting methods in code

OpenAI FLOPs per token (Python code)

```python
def openai_flops_per_token(n_layers, n_heads, d_model, n_ctx, n_vocab, ff_ratio=4):
    """Open AI method for forward pass FLOPs counting of decoder-only Transformer
    """
    d_attn = d_model // n_heads
    d_ff = d_model * ff_ratio

    embeddings = 4 * d_model
    attn_qkv = 2 * n_layers * d_model * 3 * (d_attn * n_heads)
    attn_mask = 2 * n_layers * n_ctx * (d_attn * n_heads)
    attn_project = 2 * n_layers * (d_attn * n_heads) * d_model
    ff = 2 * n_layers * 2 * d_model * d_ff
    logits = 2 * d_model * n_vocab

    return embeddings + attn_qkv + attn_mask + attn_project + ff + logits
```

DeepMind FLOPs per sequence (Python code)

```python
def deepmind_flops_per_sequence(n_layers, n_heads, d_model, n_ctx, n_vocab, ff_ratio=4):
    """DeepMind method for forwad pass FLOPs counting of decoder-only Transformer
    """
    d_attn = d_model // n_heads
    d_ff = d_model * ff_ratio

    embeddings = 2 * n_ctx * n_vocab * d_model

    attn_qkv = 2 * n_ctx * 3 * d_model * (d_attn * n_heads)
    attn_logits = 2 * n_ctx * n_ctx * (d_attn * n_heads)
    attn_softmax = 3 * n_heads * n_ctx * n_ctx
    attn_reduce = 2 * n_ctx * n_ctx * (d_attn * n_heads)
    attn_project = 2 * n_ctx * (d_attn * n_heads) * d_model
    total_attn = attn_qkv + attn_logits + attn_softmax + attn_reduce + attn_project

    ff = 2 * n_ctx * (d_model * d_ff + d_model * d_ff)

    logits = 2 * n_ctx * d_model * n_vocab

    return embeddings + n_layers * (total_attn + ff) + logits
```

Table A1: Vision Transformer FLOPs per Image

Operation	**FLOPs per Image**
Embeddings	$2\, n_{\text{patches}}\, d_{\text{patch}}\, d_{\text{patch}}\, n_{\text{channels}}\, d_{\text{model}}$
Attention: QKV	$2\, n_{\text{ctx}}\, 3\, d_{\text{model}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: QK logits	$2\, n_{\text{ctx}}\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: Softmax	$3\, n_{\text{heads}}\, n_{\text{ctx}}\, n_{\text{ctx}}$
Attention: Reduction	$2\, n_{\text{ctx}}\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right)$
Attention: Project	$2\, n_{\text{ctx}} \left(d_{\text{key}}\, n_{\text{heads}}\right) d_{\text{model}}$
Feedforward	$4\, n_{\text{ctx}} \left(d_{\text{model}}\, d_{\text{ff}}\right)$
Logits	$2\, d_{\text{model}}\, n_{\text{classes}}$
Total	Embeddings $+\ n_{\text{layers}} \times$ (Attention $+$ Feedforward) $+$ Logits

C FLOPs counting in Vision Transformer (ViT)

Extending these methods to a standard ViT is straight forward and the main difference we have to account for is the patch embeddings and logits (highlighted below). Using the DeepMind method for FLOPs counting, we can modify it as such for a ViT with a classification head in Table A1.

- n_{patches} is the number of patches in our image.
- d_{patch} is the length of the side of a patch in pixels.
  - For example, $d_{\text{patch}} = 16$ means one patch is of size $16\,\text{px} \times 16\,\text{px}$.
- n_{channels} is the number of channels in the input image.
- $n_{\text{ctx}} = n_{\text{patches}} + 1$ to account for the prepending of a learnable $[CLS]$ token.
- `Logits` is a linear layer for predicting n_{classes} using a single token as input (e.g., the $[CLS]$ token, mean pool of image tokens, etc.).

For example, if our input image is an RGB image of size $224\text{px} \times 224\text{px}$ and we have non-overlapping patches of size $16\text{px} \times 16\text{px}$ then $n_{\text{patches}} = 196$ and $n_{\text{ctx}} = 197$.

Since the patch embedding layer is applied to non-overlapping patches, we can also easily express it in terms of the total number of pixels in the image

$$C_{\text{embeddings}} = 2n_{\text{pixels}}n_{\text{channels}}d_{\text{model}}$$