

# Scaling ViTs across Training Compute

by Marvin Mboya [in](#)

## A journey across optimization levels

Looking back at when we could only reliably produce Shakespearean poetry with RNNs, a thin line between hallucinations and poetry, one can see why Google open sourcing Transformers was the just needed *krabby patty secret formula* to SOTA models toppling leaderboards every coming week, and copyright lawsuits enriching the lawyers in the same way that AI ideas could be well thought out as a well pipelined autocomplete service driving some startups. This article is a no exception, *thanks Transformers!*, written from the curiosity that inspires I to sit on the shoulders of giants, intellectually speaking, and start off this chain of optimization across languages and hardware stack that only climaxes limited to the largest GPU compute I can access without feeling like I have leaked my AWS cloud keys to the best crypto miners in the east continents!

## Back in time

Vaswani et al. didn't understand the gravity of their research<sup>1</sup> when they lightly ended their paper, but it inspired to generalize learning in the natural language domain, being largely parallelizable and solving saturation in training performance for increased training data.

Recurrent Neural Networks<sup>2</sup> was the precursor to this, its encoder that generates the latent space representation of the input tokens working in such a way that it captures the entire meaning of the input sentence in its final hidden state. This processing of the entire input text was its drawback as it could not access intermediate hidden states hence not capturing dependencies within words in the sentence.

## Sweet sauce of Transformers

Parallelizability, scaled dot product attention, and scaling of models to unprecedented size while maintaining trainability.

<sup>1</sup> [arXiv:1706.03762](#)

Attention is all you need  
Vaswani et al. 2017

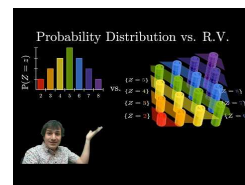
<sup>2</sup> RNNs can be understood using a special key word, **recurrent**, meaning to recur, where each hidden state would have a loop within itself and also includes the compounded outputs of all the previous hidden states, hugely based on the concept of a Markov model

**Markov process** is a stochastic process with the properties:

- number of possible states is finite
- outcome at any state depends only on outcomes of previous states
- probabilities are constant over time

where a **stochastic process** can be said as a probability distribution over a space of paths; this path often describing the evolution of some **random variable** over time.

A random variable, despite its name, is never random, and not a variable, it is a deterministic function.



Thanks to Dr Mihai for this awesome video explaining much on this

<https://youtu.be/KQHfOZHNZ3k?si=jWPeMLZV0EF76mGz>

### From a black box approach

Given a text *The ruler of a kingdom is a* with the next likely word being *king*, humanly thinking, how is the input sentence then passed to a Transformers model?

Basically, computational models cannot process strings, hence it needs conversion to a vector of integers, each word (or subword) uniquely mapped to a corresponding integer, a process known as *tokenization*. A basic form would be a hashmap of words to integers and vice versa for getting a word from index of maximum probability in softmaxed one-dimensional distribution of output float values<sup>3</sup>.

Implementing a simple tokenizer based on the vocabulary<sup>4</sup> we have,

```
text = "The ruler of a kingdom is a"
text = text.lower() # making tokenizer case insensitive
text = text.split() # getting individual words
# as separated by spaces
vocab = list(sorted(set(text)))
words_to_ids = {word:i for i, word in enumerate(vocab)}
ids_to_words = {v:k for k,v in words_to_ids.items()}
```

Great, now we have lookup tables (the last two lines), and a naive preprocessing of text needed before tokenization. So then, let's tokenize *the kingdom had another ruler*. Wait?! The lookup table does not have the words "*another*", "*had*", "*another*"! Let's improve it so any word not part of the original vocabulary be assigned a new unique id<sup>5</sup>.

```
words_to_ids = {word:i for i, word in enumerate(vocab)}
ids_to_words = {v:k for k,v in words_to_ids.items()}
def lookup(word):
    try:
        id = words_to_ids[word]
    except KeyError:
        vocab.append(word)
        words_to_ids[word] = len(vocab) - 1
        ids_to_words[len(vocab)-1] = word
        id = words_to_ids[word]
    return id
```

<sup>3</sup> the commonly used tokenizer is tiktoken, using a concept called Byte-Pair Encoding to map subwords to ids using a look-up table that takes into account frequencies of subwords.

<sup>4</sup> vocabulary ~ set of unique words (or subwords based on the tokenization strategy) in all words of the entire training dataset used to train a particular large language model.

<sup>5</sup> our look-up tables are very much capable of any encoding and decoding (for the tiny vocabulary).

## Trying our shiny code

```
sentence = "the kingdom had another ruler"
tokens = [lookup(word) for word in
          sentence.lower().split()]
print(tokens)
# [5, 2, 6, 7, 4]
words_gotten = [ids_to_words[id] for id in tokens]
sentence_gotten = " ".join(words_gotten)
print(sentence_gotten)
# "the kingdom had another ruler"
```

*Note* that the above implementation of tokenization is to help you understand a baseline of what happens under the hood in conversion of what models cannot deal with, strings, to a format that can be computationally crunched.

However, when looking into the Transformers model architecture as outlined in the paper<sup>1</sup>, also in<sup>6</sup> for convenience, it is seen that the first block is an Embedding block.

### What about the Embeddings block?

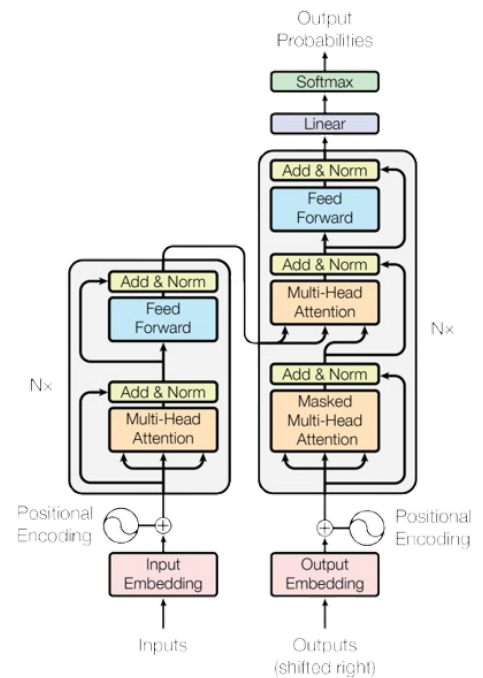
Well, the vector of integers as input in itself cannot capture rich latent representations of the input tokens, so the Embeddings block<sup>7</sup> does just that, mapping the tokens to higher dimensions. The embeddings block is usually  $V$  by  $D$ , where  $V$  is the size of the vocabulary, and  $D$  is an abstract dimension of your choosing, the higher the better, but more computationally expensive and longer to process.

Using PyTorch, an Embeddings block of  $D$  being 3 can be implemented as:

```
import torch, torch.nn as nn
V, D = len(vocab), 3
emb = nn.Embedding(V,D)
higher_emb_tokens = emb(torch.tensor(tokens))
print(higher_emb_tokens.shape) # torch.Size([5, 3])
```

One of the best LLMs ever open sourced by Meta, the Llama 3, the 3 billion parameter size variant, has its vocabulary with 128K tokens. and the embedding dimensions,  $D$ , being 3072.

## <sup>6</sup> Transformers architecture



<sup>7</sup> `nn.Embedding` is just `nn.Linear` but only that `nn.Embedding` simplifies retrieving rows from its weights such that you don't pass it one-hot vectors but just indices basically same as the position of the single 1s in the one-hot vector you would have passed to `nn.Linear`

## Positional Encoding

Before the Multi-Head Attention (MHA) block, the positional encoding is attached to the graph to constitute the position information and this allows the model to easily attend to relative positions. Why is that? Well, the MHA block is permutation-equivariant, and cannot distinguish whether an input comes before another one in the sequence or not.

The meaning of a sentence can change if words are reordered, so this technique retains information about the order of the words in a sequence.

Positional encoding is the scheme through which the knowledge of the order of objects in a sequence is maintained.

This post by Christopher<sup>8</sup> highlights the evolution of positional encoding in transformer models, a worthy read! For this article, let's focus on the rotary positional embedding (RoPE)<sup>9</sup>.

Let's making a few things clear,

- previous position encodings were done before the MHA block, this is done within it.
- RoPE is only applied to the queries and the keys, not the values.
- RoPE is only applied after the vectors  $\vec{q}$  and  $\vec{k}$  have been multiplied by the  $W$  matrix in the attention mechanism, while in the vanilla transformer they're applied before.

The general form of the proposed approach for RoPE is as in page 5 for a sparse matrix with pre-defined parameters

$$\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \dots, d/2]\}$$

which can be implemented in code as

```
assert d % 2 == 0, "dim must be divisible by 2"
i_s = torch.arange(0,d,2).float()
theta_s = 10000 ** (- i_s / d).to(device)
```

where *device* is code that chooses the compute device.

```
device = torch.device(
    "cuda" if torch.cuda.is_available() else (
        "mps" if torch.backends.mps.is_available() else "cpu"
    )
)
```

<sup>8</sup> <https://huggingface.co/blog/designing-positional-encoding>

*You could have designed state of the art positional encoding*  
Christopher Fleetwood

<sup>9</sup> [arXiv:2104.09864](https://arxiv.org/abs/2104.09864)

*RoFormer: Enhanced Transformer with Rotary Position Embedding*  
Su et al. 2022

Given the computational efficient realization which is what we're aiming at getting

<sup>10</sup> `context_len` is an integer which refers to the maximum number of tokens the model can consider in a single forward pass

$$R_{\Theta, m}^d \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

Having implemented  $\vec{\theta}$ , next let's implement  $m\vec{\theta}$  by way of an outer product<sup>10</sup>

```
m = torch.arange(context_len, device=device)
freqs = torch.outer(m, theta_s).float()
```

$$m\vec{\theta} = \text{freqs} = \begin{pmatrix} m_1\theta_1, m_1\theta_2, \dots, m_1\theta_{d/2-1}, m_1\theta_{d/2} \\ m_2\theta_1, m_2\theta_2, \dots, m_2\theta_{d/2-1}, m_2\theta_{d/2} \\ \vdots \quad \vdots \quad \dots \quad \vdots \quad \vdots \\ m_{\text{ctx\_len}}\theta_1, m_{\text{ctx\_len}}\theta_2, \dots, m_{\text{ctx\_len}}\theta_{d/2-1}, m_{\text{ctx\_len}}\theta_{d/2} \end{pmatrix}$$

It is then needed to get the complex numbers for the resulting matrix of size context len by  $d/2$ .

```
freqs_complex = torch.polar(torch.ones_like(freqs), freqs)
```

which then gives the polar form of each element in the matrix, such that

$$e^{im\vec{\theta}} = \begin{pmatrix} \cos(m_1\theta_1) + i \sin(m_1\theta_1), \cos(m_1\theta_2) + i \sin(m_1\theta_2), \dots, \cos(m_1\theta_{d/2}) + i \sin(m_1\theta_{d/2}) \\ \cos(m_2\theta_1) + i \sin(m_2\theta_1), \cos(m_2\theta_2) + i \sin(m_2\theta_2), \dots, \cos(m_2\theta_{d/2}) + i \sin(m_2\theta_{d/2}) \\ \vdots \quad \quad \quad \vdots \quad \quad \quad \dots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ \cos(m_{cl}\theta_1) + i \sin(m_{cl}\theta_1), \cos(m_{cl}\theta_2) + i \sin(m_{cl}\theta_2), \dots, \cos(m_{cl}\theta_{d/2}) + i \sin(m_{cl}\theta_{d/2}) \end{pmatrix}$$

Let's consider a subset of the inputs and a subset of the matrix above, then

$$\begin{aligned} \vec{x} &= \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} = \begin{pmatrix} x_1 + ix_2 \\ x_3 + ix_4 \end{pmatrix} \otimes \begin{pmatrix} f_{11} + i\hat{f}_{11} \\ f_{12} + i\hat{f}_{12} \end{pmatrix}, \text{ where } \begin{cases} f_{11} = \cos(m_1\theta_1) \\ \hat{f}_{11} = \sin(m_1\theta_1) \\ f_{12} = \cos(m_1\theta_2) \\ \hat{f}_{12} = \sin(m_1\theta_2) \end{cases} \\ &= (x_1 + ix_2)(f_{11} + i\hat{f}_{11}) = x_1f_{11} - x_2\hat{f}_{11} + i(x_1\hat{f}_{11} + x_2f_{11}) \\ &\quad \text{meaning } \begin{pmatrix} x_1 + ix_2 \\ x_3 + ix_4 \end{pmatrix} \otimes \begin{pmatrix} f_{11} + i\hat{f}_{11} \\ f_{12} + i\hat{f}_{12} \end{pmatrix} \\ &= \begin{pmatrix} x_1f_{11} - x_2\hat{f}_{11} + i(x_1\hat{f}_{11} + x_2f_{11}) \\ x_3f_{12} - x_4\hat{f}_{12} + i(x_3\hat{f}_{12} + x_4f_{12}) \end{pmatrix} = \begin{pmatrix} x_1f_{11} - x_2\hat{f}_{11} & x_1\hat{f}_{11} + x_2f_{11} \\ x_3f_{12} - x_4\hat{f}_{12} & x_3\hat{f}_{12} + x_4f_{12} \end{pmatrix} \\ &\quad \text{rearranging gives} \\ &= \begin{pmatrix} x_1f_{11} - x_2\hat{f}_{11} \\ x_1\hat{f}_{11} + x_2f_{11} \\ x_3f_{12} - x_4\hat{f}_{12} \\ x_3\hat{f}_{12} + x_4f_{12} \end{pmatrix} \Rightarrow \begin{pmatrix} x_1 \cos m_1\theta_1 - x_2 \sin m_1\theta_1 \\ x_1 \sin m_1\theta_1 + x_2 \cos m_1\theta_1 \\ x_3 \cos m_1\theta_2 - x_4 \sin m_1\theta_2 \\ x_3 \sin m_1\theta_2 + x_4 \cos m_1\theta_2 \end{pmatrix} \end{aligned}$$

## Implementing the rotation mechanism

the previously derived mathematical algorithm can then be translated into code as below.

```
def apply_rotary_embs(x, freqs_complex, device):
    # x rearrange and make complex => result => x1 + jx2
    # [B, context_len, H, head_dim] => [B, context_len, H, head_dim/2]
    x_c = torch.view_as_complex(
        x.float().reshape(*x.shape[:-1], -1, 2)
    )
    # [context_len, head_dim/2] => [1, context_len, 1, head_dim/2]
    f_c = freqs_complex.unsqueeze(0).unsqueeze(2)
    # [B, context_len, H, head_dim/2] * [1, context_len, 1, head_dim/2]
    # => [B, context_len, H, head_dim/2]
    x_rotated = x_c * f_c
    # [B, context_len, H, head_dim/2] => [B, context_len, H,
        head_dim/2, 2]
    x_out = torch.view_as_real(x_rotated)
    # [B, context_len, H, head_dim/2, 2] => [B, context_len, H,
        head_dim]
    x_out = x_out.reshape(*x.shape)
    return x_out.type_as(x).to(device)
```

And now to the most interesting part of this architecture....

## Multi-Head Attention<sup>13</sup>

a picture is worth a thousand words! Let it do the talking!

<sup>11</sup> `nn.Linear` is an instance initialization of a stack of perceptrons in a single layer in PyTorch, with `d_in` previously known as the abstract dim of the word embedding, and `d_out` is initialized as `d_in`

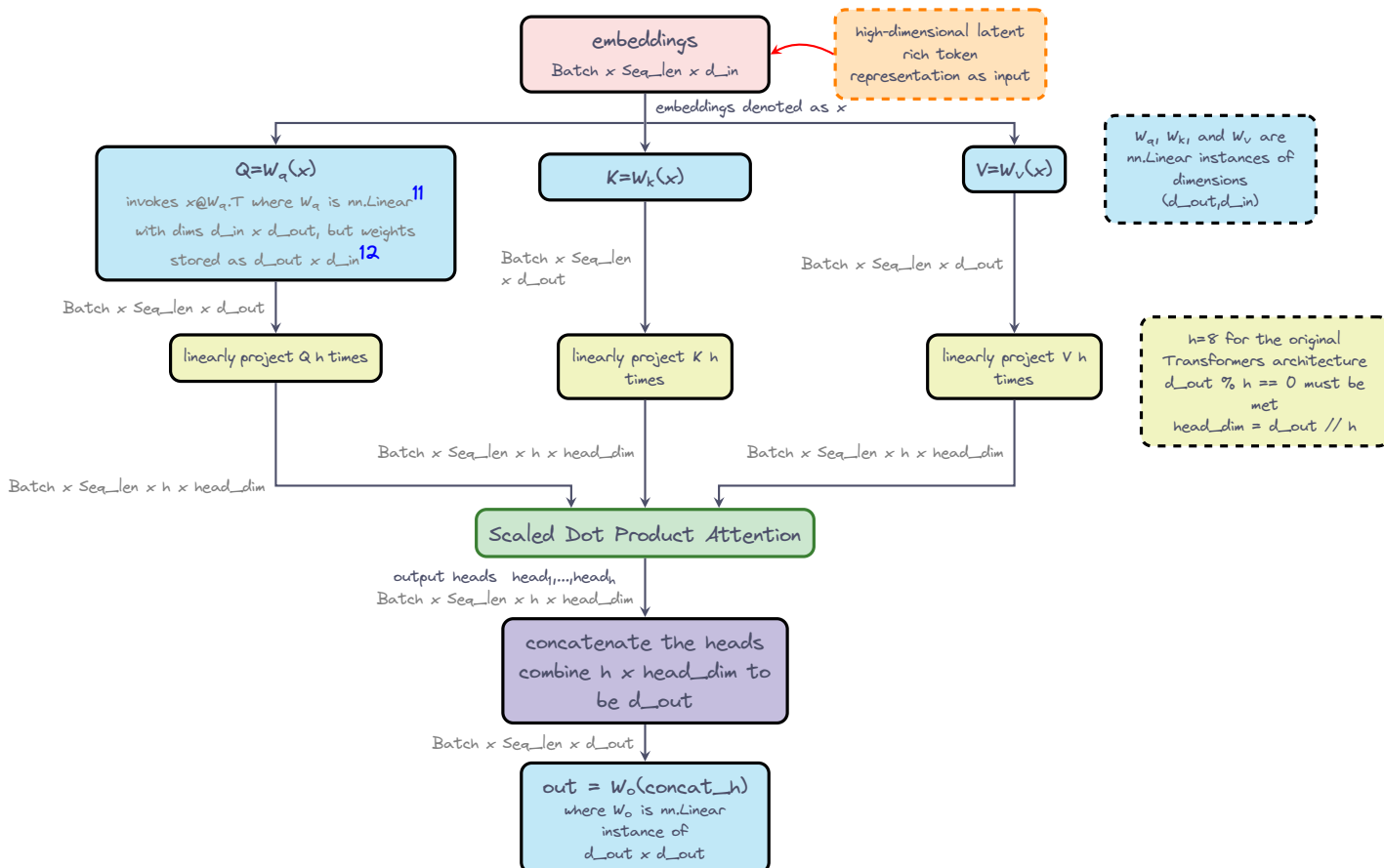
<sup>12</sup> proving the invocation that initializes Q, K and V matrices

```
import torch
import torch.nn as nn
x = torch.randn(10, 3)
Wq = torch.nn.Linear(3, 40, bias=False)
torch.equal(Wq(x), x.dot(Wq.weight.T))
torch.equal(Wq(x), x@Wq.weight.T) # True
```

<sup>13</sup> the MHA has its core in attention mechanism whose goal is to dynamically decide on which inputs we want to “attend” more than others based on

- *query* ~ a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- *keys* ~ for each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.

...



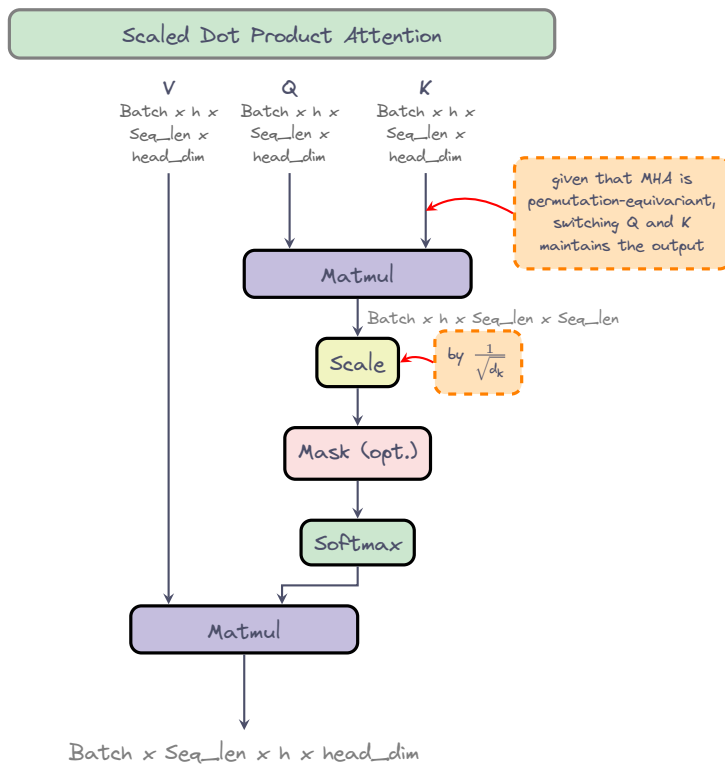
## Scaled dot product attention

The term, first introduced in the *Vaswani et al.* paper, involves the following key operations:

- compute the dot product of queries and keys of dimension  $d_k$ ,  $QK^T$
- scaling by a factor  $1/\sqrt{d_k}$  to counteract the effect of extremely small gradients in the softmax computation as will be seen in the next step when  $d_k$  becomes very large<sup>14</sup>. This begets the attention scores.
- softmax computation of the normalized result attention scores. The result is the attention weights.
- dot product of the attention weights and the values.

the infamous equation is therefore

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



From the diagram above, there's a new block, **Mask**, that does something called masking. A transformer usually has two phases, encoding phase and the decoding phase. From the Transformers architecture diagram, encoder is on the left and the decoder on the right for the two phases.

from previous<sup>13</sup>...

- **values**  $\sim$  for each input element, we also have a value vector. This feature vector is the one we want to average over.
- **score function**  $\sim$  to rate which elements we want to pay attention to, we need to specify a score function. The score function takes the query and a key as input, and outputs the score (attention weight) of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.



courtesy of [UvA course notes](#)

<sup>14</sup>  $d_k$  is the size of the last dimension of the keys after linear projection and transpose, to be implemented later. It is the head dimension for each attention head. Sanity check states that your key dimension be  $B \times \text{Seq\_len} \times h \times \text{head\_dim}$  before this step where  $d_k$  is gotten by  $k.\text{shape}[-1]$

During the decoding phase, at each step of predicting a word<sup>15</sup>, the network needs take a look at the words previous to that step, and output a softmax prediction for what it thinks the next word is. Since transformers attend to the entire sequence, before and after, it becomes a trivial task to predict the next word, simply by putting 100% attention to the word after it.

This of course is cheating, it won't learn anything really. During the inference pipeline, the entire sequence won't be present, hence why we need the masking block, we don't want each word in the decoder to see the words that come after it.

### Implementing masking in code

Let's use the sequence below

*Eiffel Tower is in Paris*

and consider the llama 2 tokenizer<sup>16</sup>, *sentencepiece*, as the final Transformers model built on these progressive learnings while building on the architecture is Llama 2.

```
import sentencepiece as spm
sequence = "Eiffel Tower is in Paris"
sp = spm.SentencePieceProcessor("llama-2-7b-tok.model")
tokens = sp.encode_as_ids(sequence)
```

Considering V and D used for *Llama2 model 7B* variant, let's initialize an embedding instance.

```
V,D=32_000,4_096
emb = nn.Embedding(V, D)
emb_tokens = emb(torch.tensor(tokens))
print(emb_tokens.shape)
# torch.Size([7, 4096])
```

Our embeddings output being the input to scaled dot product attention, let's compute  $QK^T$  then scale keeping in mind that the batch dimension, multiple heads, and the positional encoding is not incorporated for the sake of focusing on masking.

```
Wq, Wk, Wv = nn.Linear(D,D),nn.Linear(D,D),nn.Linear(D,D)
q, k, v = Wq(emb_tokens), Wk(emb_tokens), Wv(emb_tokens)
scores=q@k.T
scaled_scores=scores/k.shape[-1]**.5
print(scaled_scores.shape) # torch.Size([7, 7])
```

<sup>15</sup> the model actually predicts a token which, by using a lookup table, is decoded to a word which is what humans understand.

<sup>16</sup> the lookup-table *tokenizer.model* can be found from the huggingface model card for *Llama-2-7b* <https://huggingface.co/meta-llama/Llama-2-7b/tree/main>



```
torch.set_printoptions(precision=5, sci_mode=False, linewidth=500)
print(scaled_scores)
tensor([[ -0.10457, -0.23802,  0.08053,  0.33000, -0.10408,  0.55068,  0.68916],
        [ -0.35013, -0.04846,  0.65688,  0.18756, -0.81784,  0.10682, -0.74313],
        [ -0.26961, -0.70423,  0.94224,  0.16090, -0.20169,  0.15549, -0.28134],
        [ -0.32253,  0.56740,  0.08793, -0.53429, -0.19362, -0.22245, -0.38808],
        [  0.32020,  0.29380,  0.18501, -0.53281,  0.02592, -0.57664,  0.17737],
        [  0.00706, -0.08485, -0.11895,  0.21021,  0.50643,  0.48187,  0.11625],
        [  0.38275,  0.45847, -0.34459, -0.12443,  0.35930,  0.65530,  0.03805]],
        grad_fn=<DivBackward0>)
```

Now onto a mask with ones from the first upper off-diagonal onwards. Then, fill them with  $-\infty$  such that the exponential of those values will be zero in the weights.

```
mask = torch.triu(torch.ones_like(scaled_scores),
                  diagonal=1)
scaled_scores_masked =
    scaled_scores.masked_fill_(mask.bool(), -torch.inf)
weights = torch.softmax(scaled_scores_masked, dim=-1)
```

Now, for the weights, pre-matrix multiply with V for the result of Scaled Dot Product Attention

```
out = weights @ v
print(out.shape) # torch.Size([7, 4096])
```

Nice! Now onto *Add & Norm* layer, which from the paper, is a Layer normalization that computes

$$\text{LayerNorm}(x + \text{Multihead}(x))$$

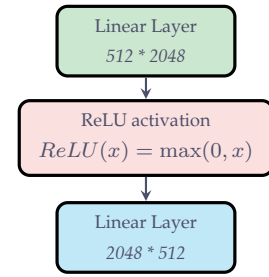
where  $x$  is basically the same sequence (as an embedding) input to the  $Q, K \& V$ . This layer hence is a residual connection necessary for enabling smooth gradient flow through the model and retaining information from the original sequence prior to the multi-head attention. This is simply implemented as

```
out_attn = multiheadAttn(x)
out = x + out_attn
norm_out = norm(out)
```

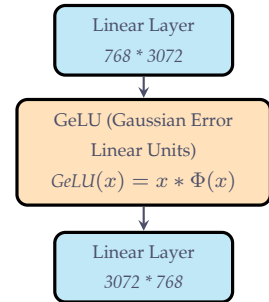
### What about the Feed Forward Network layer

Always forming a crucial layer in most models, the FFN, in this case, maps context rich vectors onto a higher dimension<sup>17</sup> which increases learning so it can model more complex relationships and also adds an activation function to introduce non-linear, even better relations.

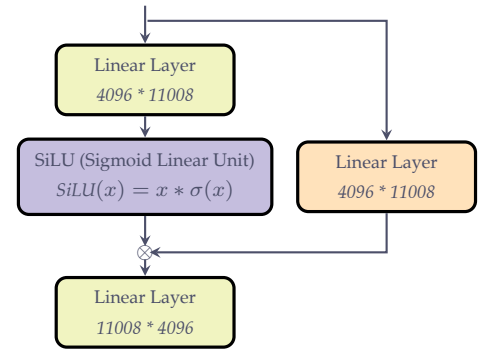
#### <sup>17</sup> Feed Forward NN layer for Transformer model



#### Feed Forward NN layer for GPT-2



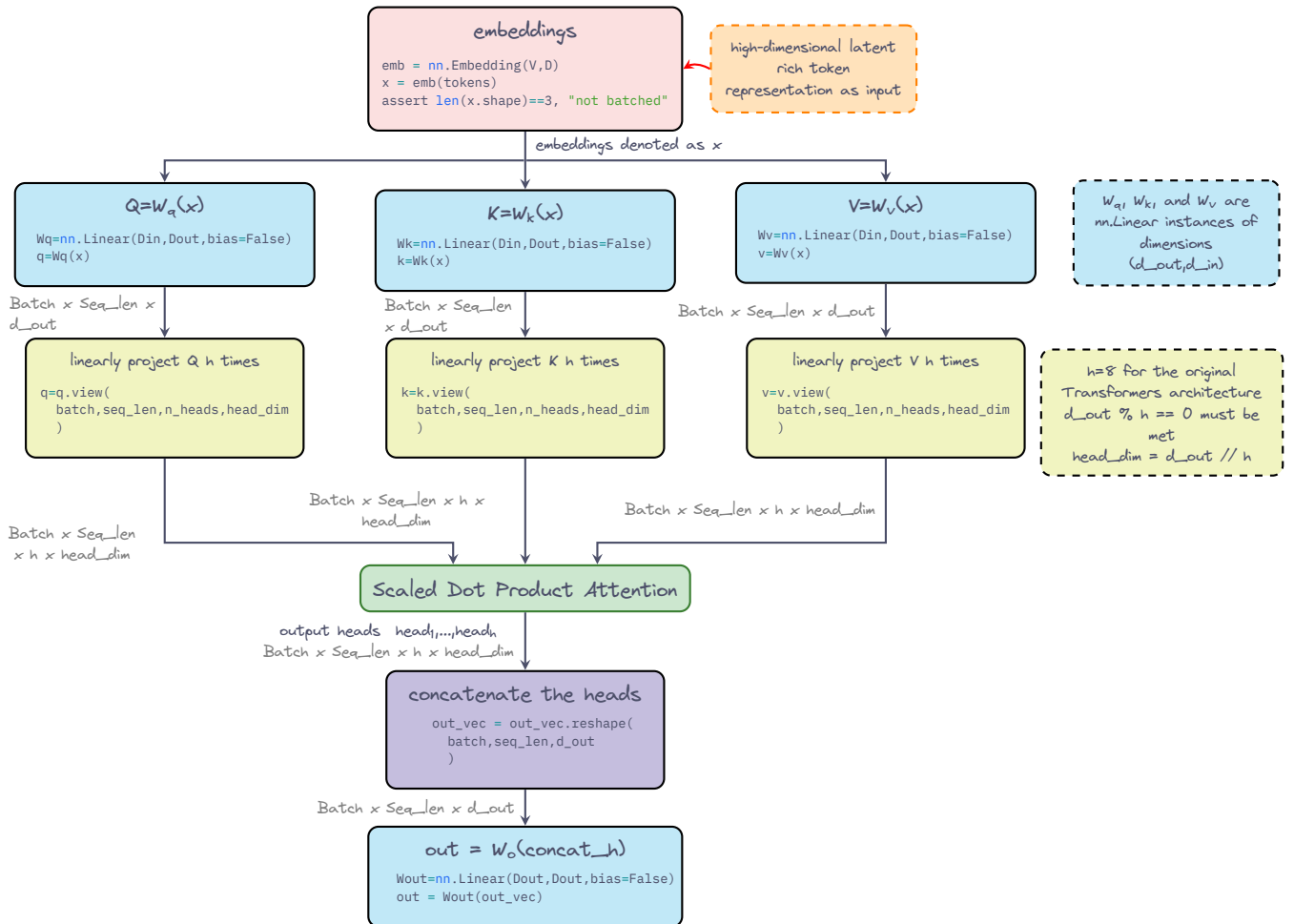
#### Feed Forward NN layer for Llama-2-7b



## Building Llama-2 from the SDPA outwards...

Gladly having gone through the layers in the Transformer model, it is of essence to build the Llama-2 model graph and load the weights for the 7B variant. It is a decoder-only architecture, as is most State of The Art common LLMs. Why is that? Well, decoder-only architectures worked very well for next token prediction and translation tasks, and were easier to train. And so, they picked up as the *de facto* baselines for most current outstanding models.

Earlier, we had the graph for the Multi-head Attention, let's add codes to it to map it to implementation.



But wait! what about the RoPE implementation, remember that as has been discussed earlier, positional encodings should be somewhere in the above disjointed<sup>18</sup> graph of a code. Let's figure out where?

## Recap on Rotational Positional Encoding

```
def precompute_freqs_cis(d, context_len, theta =
10_000, device = "gpu"):
    #
    #
    assert d % 2 == 0, "dim must be divisible by 2"
    #
    i_s = torch.arange(0,d,2)[: (d//2)].float()
    theta_s = theta ** (- i_s / d).to(device)
    m = torch.arange(context_len, device=device)
    freqs = torch.outer(m, theta_s).float()
    freqs_cis = torch.polar(torch.ones_like(freqs),
        freqs)
    return freqs_cis
```

<sup>19</sup> reminder that the rotational transformation is to be applied to the queries and keys only and not the values (refer to page 4).

As the paper <sup>9</sup> says, "...to any  $x_i \in \mathbb{R}^d$  where  $d$  is even..."

$i_s = 2(i-1)$  for  $i \in \{1, 2, \dots, d/2\}$

$10000^{-i_s/d}$  which expands to  $10000^{-2(i-1)/d}$

outer product of  $\vec{m}$  &  $\vec{\theta}$  to give

$$\begin{pmatrix} m_1\theta_1 & m_1\theta_2 & \dots & m_1\theta_{d/2-1} & m_1\theta_{d/2} \\ m_2\theta_1 & m_2\theta_2 & \dots & m_2\theta_{d/2-1} & m_2\theta_{d/2} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ m_d\theta_1 & m_d\theta_2 & \dots & m_d\theta_{d/2-1} & m_d\theta_{d/2} \end{pmatrix}$$

elementwise mapping i.e.  
 $m_1\theta_1 \Rightarrow \cos(m_1\theta_1) + i \sin(m_1\theta_1)$   
where the ones are the absolute value arguments

takes each group of 2s of elements, ...  
[x, y],  
[m, n], ...  
to single elements of  
 $x+yj$ ,  
 $m+jn$ ...

```
def apply_rotary_embs(x, freqs_cis, device):
    #
    #
    x_c = torch.view_as_complex(
        x.float().reshape(*x.shape[:-1], -1, 2)
    )
    #
    #
    f_c = freqs_cis.unsqueeze(0).unsqueeze(2)
    #
    #
    x_rotated = x_c * f_c
    #
    x_out = torch.view_as_real(x_rotated)
    #
    x_out = x_out.reshape(*x.shape)
    return x_out.type_as(x).to(device)
```

dynamically expands the last dimension  
 $(\dots, d1)$  to  $(\dots, \frac{d1}{2}, 2)$  where  $d1$  is even

dims transformed from  $(\dots, d)$  to  $(\dots, \frac{d}{2})$

reverses the effect of torch.view\_as\_complex

With the knowledge of the implementation of the rotational positional encodings, let's inject it into the graph for the MultiHead Attention after the transformation

$$[batch \times seq\_len \times n\_heads \times head\_dim]$$

but before the high-dimensional transpose to get the batch of heads each with dimensions  $(seq\_len, head\_dim)$ <sup>19</sup>.

★ which is then done below<sup>20</sup>

```
# Already defined earlier
dim=4096; n_heads=32; context_len=4096
Q,K,V=... # each dims being (Batch,SeqLen,Heads,HDim)
m_theta_polar_tensor =
    precompute_freqs_cis(dim//n_heads,
        context_len*2,"cpu")
m_theta_polar_seq = m_theta_polar_tensor[:seq_len]
Q=apply_rotary_emb(Q,m_theta_polar_seq)
K=apply_rotary_emb(K,m_theta_polar_seq)
```

<sup>20</sup> full neat implementation

[https://github.com/Marvin-desmond/ScalingViTsAcrossTrainingCompute/blob/main/mha/mha\\_with\\_rope.py](https://github.com/Marvin-desmond/ScalingViTsAcrossTrainingCompute/blob/main/mha/mha_with_rope.py)

Llama 2 Multi-Head Attention with ROPE

### Unwrapping the Transformer Block

As much as the original Transformer does the normalization as

$$\text{LayerNorm}(x + \text{Multihead}(x))$$

Llama2 does a prenormalization given by

$$x_n = \text{RMSNorm}(x)$$

$$\text{out} = x + \text{Multihead}(x_n)$$

where

$$\text{RMSNorm}(x) = \frac{x_i}{\text{RMS}(x)} * \gamma_i$$

$$\text{RMS}(x) = \sqrt{\epsilon + \frac{1}{n} \sum_{i=1}^n x_i^2}$$

which works out in code as

```
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        means = x.pow(2).mean(-1, keepdim=True)
        norm_x = x * torch.rsqrt(means + self.eps)
        return (norm_x * self.weight).to(x.dtype)

rmsNorm=RMSNorm(dim) # dim=4096
x_norm=rmsNorm(x) # x => embeddings => (Batch,SeqLen,Dim)
# some mhAttention already instantiated called below
attn_out=mhAttention(x_norm)
# then add
out = x + attn_out
```

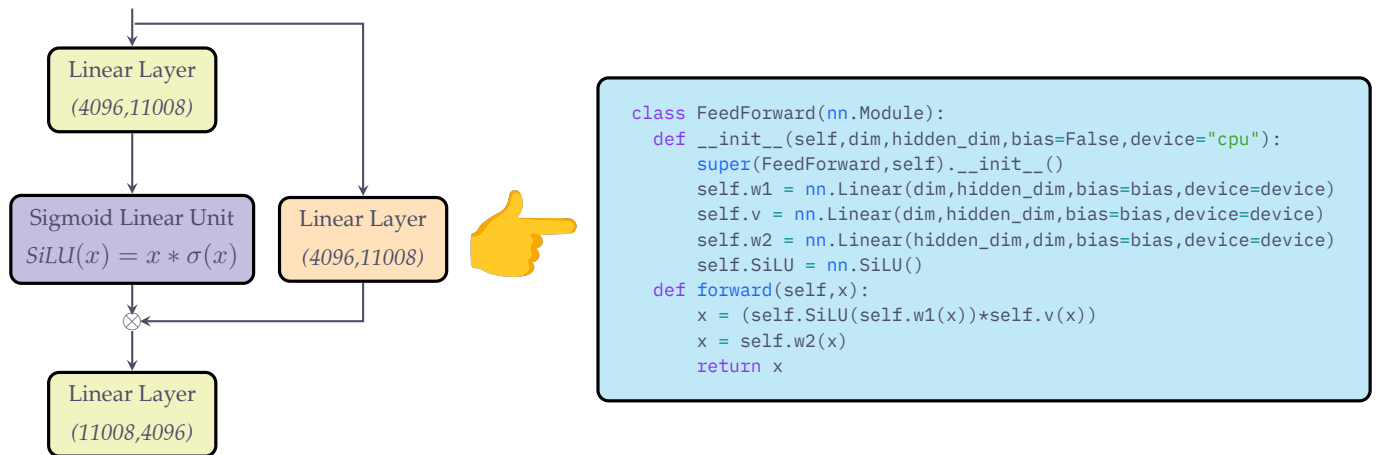
with the pre-normalization done to the input to the attention block and to the input to the feed-forward networks.

However, the original FFN, as can be seen from the side notes on pg.9, does two linear transformations with a ReLU<sup>21</sup> activation function applied between the two linear transformations.

$$FFN(x, W_1, W_2, b_1, b_2) = \max(0, xW_1 + b_1)W_2 + b_2$$

the above equation being representative of the graph computation in the linear topology on the just aforementioned page.

Llama2, the current LLM architecture of interest in implementation in this section of the article, focuses on a Linear Unit known as SwiGLU<sup>22</sup>, a variation of the Transformer FFN layer which then uses a variant of the Gated Linear Unit<sup>23</sup>. This leads to the FFN layer having three weight matrices as opposed to the original two which yields the implementation below.



With the  $\star$  operation being the Hadamard product, or as commonly known, the elementwise product, of the two Weight matrices of dimensions (4096, 11008) to give a resulting matrix maintaining the given dimensions.

In the general implementation for any given Llama 2 variant, the hidden dimension size is gotten by

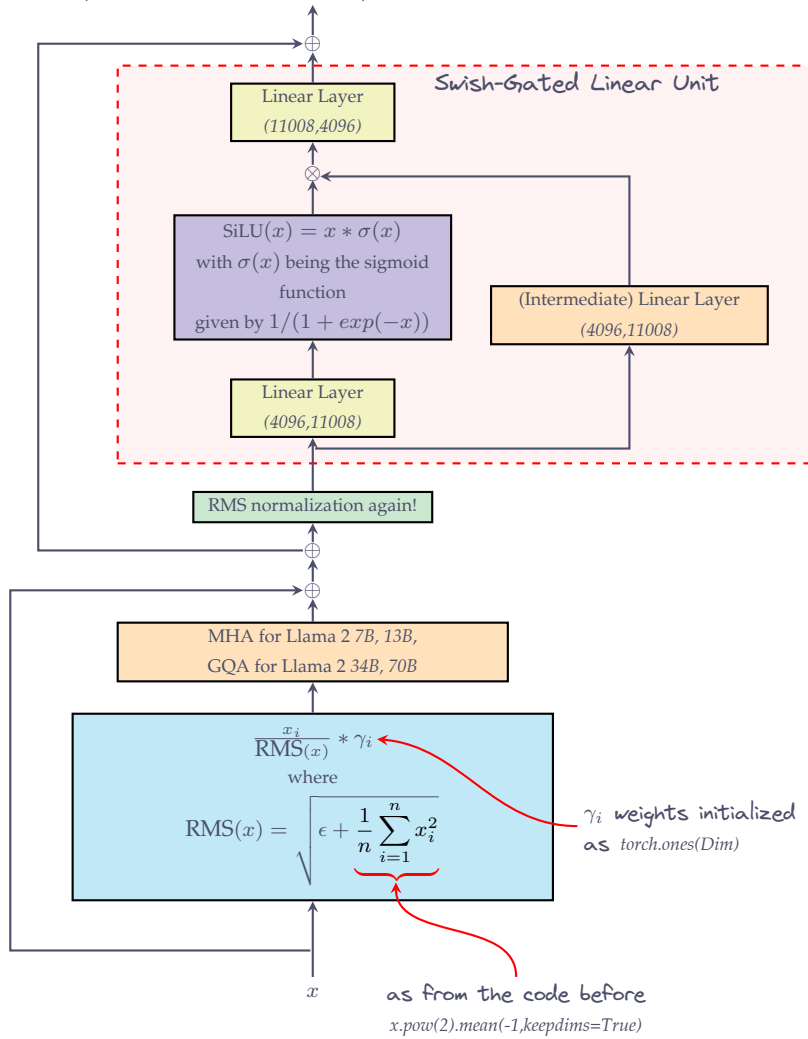
- scaling of  $dim$  by 4
- reduction by  $2/3$
- adjust it as a factor of a given multiple for computational efficiency

<sup>21</sup> <https://proceedings.mlr.press/v15/glorot11a.html>  
Deep Sparse Rectifier Neural Networks  
Glorot et al. 2011

<sup>22</sup> <https://arxiv.org/abs/2002.05202v1>  
GLU Variants Improve Transformer  
Noam Shazeer 2020

<sup>23</sup> <https://arxiv.org/abs/1606.08415>  
Gaussian Error Linear Units (GELUs)  
Dan Hendrycks, Kevin Gimpel 2016

Hence, from the clarifications, the whole Transformer block is visualized as



With the above nice input-output mapping translating to code as

```
class TransformerBlock(nn.Module):
    def __init__(self, d_in, d_out, n_heads, context_window, device="cpu"):
        super(TransformerBlock, self).__init__()
        self.rms_attn = RMSNorm(d_in, device=device)
        self.attn = MHAandRoPE(d_in, d_out, n_heads, context_window, device=device)
        self.rms_ffn = RMSNorm(d_in, device=device)
        self.ffd = FeedForward(d_in, 4*d_in, device=device)

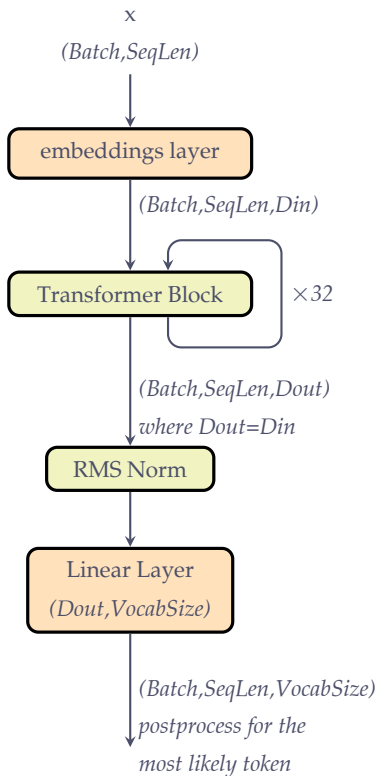
    def forward(self, x, m_thetas):
        attn_x = self.rms_attn(x)
        h = self.attn(attn_x, m_thetas) + x

        ffn_x = self.rms_ffn(h)
        out_x = self.ffd(ffn_x)
        x = out_x + h
        return x
```

## The whole Llama 2 picture

Building this architecture has been interesting, so now let's go out from the Transformer block to the whole Llama 2 model, having a full understanding of the architecture.

with the model translating to code as



for parameters

```

class CONFIG:
    VOCAB: int = 32_000
    CONTEXT_LEN: int = 4096
    DIM: int = 4096
    N_HEADS: int = 32
    N_LAYERS: int = 32
    HIDDEN_DIM: int = 11008
    DTYPE: torch.dtype =
        torch.bfloat16
  
```

```

class TransformerLlama2(nn.Module):
    def __init__(self, CONFIG: CONFIG, device="cpu"):
        super(TransformerLlama2, self).__init__()
        self.token_embeddings = nn.Embedding(
            CONFIG.VOCAB, CONFIG.DIM,
            device=device)
        self.layers = nn.ModuleList()
        for _ in range(CONFIG.N_LAYERS):
            self.layers.append(
                TransformerBlock(
                    CONFIG.DIM, CONFIG.DIM,
                    CONFIG.N_HEADS, CONFIG.CONTEXT_LEN,
                    device=device))
        self.norm = RMSNorm(CONFIG.DIM, device=device)
        self.output = nn.Linear(
            CONFIG.DIM, CONFIG.VOCAB,
            bias=False, device=device
        )
        self.m_thetas = precompute_freqs_cis(
            CONFIG.DIM//CONFIG.N_HEADS,
            CONFIG.CONTEXT_LEN*2,
            device=device
        )

    def forward(self, x):
        batch, seq_len = x.shape
        x = self.token_embeddings(x)
        m_thetas_seq = self.m_thetas[:seq_len]
        for layer in self.layers:
            x = layer(x, m_thetas_seq)
        x = self.norm(x)
        x = self.output(x).float()
        return x
  
```

The architecture now complete, the inferencing of the model given the loading of the weights is the key section to follow. However, Large Language Models and in this case Llama2, even though decoder only, is still high memory demanding and so cannot be easily run on local computes. Therefore, inferencing brings into light cloud computes that effectively run inference pipelines.

★ The current snapshot for the inference pipeline is now <https://github.com/Marvin-desmond/>

[ScalingViTsAcrossTrainingCompute/blob/main/transformerLlama2/local\\_inference.py](https://github.com/ScalingViTsAcrossTrainingCompute/blob/main/transformerLlama2/local_inference.py)

### First shot at inference

As much as we would want to look for the cloud instance on AWS with the highest GPU VRAM and spawn it, ssh to it then copy the inference files and folders to the remote instance before running the pipeline, and then worrying about destroying the instance before our expenses gets too high, let's simplify things a bit shall we! As on-demand as we can get and with just focusing on the Python code with a bit of sprinkling of decorators, I'd like to go into this platform called Modal<sup>24</sup>

### Configuring Modal

After installing Modal, you can run a python file using

```
modal run hello.py
```

instead of

```
python hello.py
```

To illustrate on the local entry point for Modal in the code, let's say the code in the file is initially

```
def func():
    import subprocess
    try:
        subprocess.run("nvidia-smi")
    except:
        print("CUDA not found")

if __name__ == "__main__":
    func()
```

To have it compatible with cloud running, we'll have to decorate *func* as shown

```
import modal
app = modal.App()

@app.function()
def func():
    import subprocess
    try:
        subprocess.run("nvidia-smi")
    except:
        print("CUDA not found")
```



the local entry point will now change from

```
if __name__ == "__main__":  
    func()
```

to now being

```
@app.local_entrypoint()  
def main():  
    func.local()  
    func.remote()
```

where the function can now be invoked on your local compute using `func.local()` and Modal's remote compute using `func.remote()`, and that's about it! For those familiar with CUDA, it is like prepending the keywords `__host__` `__device__` to a function without having to rewrite the whole function for each compute. No need to ssh or maintain any GPU instance! The results for the functions, assuming your local compute is CPU-only, will be

CUDA not found

CUDA not found

Modal runs on CPU by default for the remote compute, so let's add a GPU option<sup>25</sup>, for now going for the T4.

```
import modal  
app = modal.App()  
  
@app.function(gpu="T4")  
def func():  
    import subprocess  
    try:  
        subprocess.run("nvidia-smi")  
    except:  
        print("CUDA not found")
```

and for the result!

```
CUDA not found  
Wed May 28 20:41:22 2025  
+-----+  
| NVIDIA-SMI 570.86.15              Driver Version: 570.86.15    CUDA Version: 12.8     |  
+-----+  
| GPU   Name                               Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |  
| Fan   Temp   Perf              Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute M. |  
|                               |                      |              MIG M. |  
+-----+  
| 0   Tesla T4                               On          | 00000000:00:1C:0 Off |             0        |  
| N/A   24C    P8              9W /   70W |  1MiB / 15360MiB |      0%      Default |  
|                               |                      |              N/A     |  
+-----+  
+-----+
```

25

 T4 ~ 16GB VRAM

 L4 ~ 24GB VRAM

 A10G ~ 24GB VRAM

 A100-40GB

 A100-80GB

 L40S ~ 48GB VRAM

 H100 ~ 80GB VRAM

 H200 ~ 141GB VRAM

 B200 ~ 180GB VRAM

## Configuring the pipeline for GPU inference

Now that we have a good enough understanding of Modal, let's configure the file *local\_inference.py* for remote compute.

We'll need torch for GPU accelerated numerical computing, huggingface hub for downloading llama weights, sentencepiece as the tokenizer package for Llama2. So let's create an image that has those packages, and also upload the corresponding necessary files to remote that defines the classes and utilities for the model implementation.

```
import modal
app = modal.App("llama-gpu-inference")
image = modal.Image.debian_slim().pip_install(
    "torch", "numpy", "sentencepiece",
    "huggingface_hub[hf_transfer]"
).env({"HF_HUB_ENABLE_HF_TRANSFER": "1"})
.add_local_file(
    local_path="./core.py", remote_path="/root/core.py"
).add_local_file(
    local_path="./block_utils.py", remote_path="/root/block_utils.py"
).add_local_file(
    local_path="./pos_freqs.py", remote_path="/root/pos_freqs.py"
).add_local_dir(local_path="./mha", remote_path="/root/mha")
```

Let's then provision a Modal volume for saving the weights.

```
from pathlib import Path
volume = modal.Volume.from_name("model-weights-vol",
    create_if_missing=True)
MODEL_DIR = Path("/models") # note the dot is removed
```

Next, we make our function to download model weights run on remote compute by decorating it as follows

```
@app.function(
    volumes={MODEL_DIR: volume},
    image=image,
    secrets=[modal.Secret.from_name("huggingface-secret")])
def download_model(
    repo_id: str="meta-llama/Llama-2-7b",
    revision: str=None, # include a revision to prevent
    surprises!
):
    # more code below ...
```

and by changing the function call as<sup>26</sup>

```
download_model.remote()
```

<sup>26</sup> ensure the huggingface secret is configured since Llama weights access requires authentication, and also remove the

```
from dotenv import load_dotenv
load_dotenv()
```

and

```
if not torch.cuda.is_available():
    sys.exit(0)
```

snippets of codes from the original *local\_inference.py* code file for it to work with the remote compute.

## Choosing the right GPU

This is the core question for us to choose the GPU that fits our memory needs during inference whilst also being economical but not by reducing the reliable output of tokens/sec. We cannot use the T4 because as this equation states<sup>27</sup>, the gpu memory (in GB) denoted as  $M$  is given by

$$M = \left( \frac{P \times 4B}{32/Q} \right) \times 1.2$$

where

$P \sim$  amount of parameters in the model

$4B \sim$  4 bytes, the bytes used for each parameter

$32 \sim$  there are 32 bits in 4 bytes

$Q \sim$  amount of bits for loading the model, 16 bits, 8 bits, or 4 bits

$1.2 \sim$  20% overhead of additional loading in GPU memory

For *Llama2 model 7B*, which obviously has 7B<sup>28</sup> parameters, currently being inferenced at full precision, hence yielding  $Q$  as 32, the lower bound for GPU is then

$$M = \left( \frac{7 \times 10^9 \times 4}{32/32} \right) \times 1.2$$

$$M = 3.36 \times 10^{10} \text{ bytes}$$

$$M = 33.6 \text{ GB}$$

Hence for the GPU options by Modal, we can then go for the nearest upper GPU which is A100-40GB. This leads to decorating our class as

```
@app.cls(  
    gpu="A100-40GB",  
    volumes={MODEL_DIR: volume},  
    image=image  
)  
  
class PIPELINE:  
    # more code ...  
    device = "cuda"
```

and interesting changes to the `__init__` and the `inference` methods<sup>29</sup>.

<sup>27</sup> Calculating GPU memory for serving LLMs

<sup>28</sup> what if we didn't know the number of parameters? Well for starters, we can get the parameters of filters in convolution layers by knowing the number of filters and the number of channels per each input to that layer and the kernel size (depthwise stack of kernels form a filter). For a Linear layer, we get the size of the weight matrix and the bias to compute the parameters in that layer.

<sup>29</sup>

```
def __init__(self, device):  
    # ...
```

becomes

```
@modal.enter()  
def enter(self):  
    # ...
```

and the inference method is decorated as

```
@modal.method()  
def inference(self):  
    # ...
```

with the function call being changed to

```
@app.local_entrypoint()  
def main():  
    download_model.remote()  
    pipeline = PIPELINE()  
    pipeline.inference.remote()
```

And so trying this prompt

*The interesting life of the blue eyed child from a glass orb*  
gives

*The interesting life of the blue eyed child from a glass orb. A story of the unusual life of a young girl who grew up in the midst of the great depression. She saw a lot in her short life. It's a heart warming story of a little girl who grew into a wonderful woman. This is a biography of a woman who grew up in the south during the Great Depression, who then worked her way through college and became a successful attorney and judge. It's the story of a young girl who grows up in the midst of the depression and becomes a successful attorney. It's a great story with a lot of heart. I loved this book. It was such a great read. I loved the story of a girl growing up in the midst of the depression, and how she made her way through life. This is a wonderful story about a young girl who grew up in the midst of the Great Depression and how she was able to make it through. She was a strong and determined young woman and her story is very inspiring. I would recommend this book to anyone. This is a very interesting and inspiring story. It is the story of a young girl growing up in the midst of the Great Depression and how she was able to make it through. She was a strong and determined young woman and her story is very inspiring...*

## *From Transformers To Vision Transformers*

### *Revealing The Motivation*

The YouTube recommendation, being as tuned as ever to the videos I liked to watch, recommended me this one video<sup>30</sup>. And of course, developers from DeepMind being the first to present was an awesome start to the video, having loved the kind of impactful research DeepMind does, AlphaFold 2<sup>31</sup> being the first of many that stuck in my mind.

This video goes into optimization using Jax<sup>32</sup>, as I would call it, a framework that's so powerful at granular and complex differentiation and JiT compiles to GPU and TPUs, a very interesting combo for High Performance Computing. Imagine wanting to build ML workloads in an efficient of code as you can possibly get.

A study done, presented by one Kathleen, gives a walkthrough on the speed and cost of training a ViT<sup>33</sup> given different performance metrics quantifying how fast training gets.<sup>34</sup> This precursor study was a huge stepping stone to training Gemma group of models by Google, *Gemma 3n*<sup>35</sup> being my most beloved, given it was focused on optimization and hence inference for relatively lower memory-constrained devices.

### *The focus of the article*

With this in mind, the article will now convert the previous Llama 2 architecture to the base Transformer model for which the Google ViT is based upon, before now looking into the ViT paper on changes to achieve the final Vision Transformers state. However, we are still in the single-GPU pipeline, hence changes need be made to the architecture to shard it across many GPUs, optimizing it even by float point precision levels as we tune it to its optimal state ever.

### *So why Vision Transformers?*

Given the interesting nature of Transformers of being computationally efficient and scalable, allowing training models of unprecedented size with no sign of saturating performance, and convolutional architectures being so good at computer vision, the research on ViTs aimed at improving Transformer models for image capabilities.

<sup>30</sup> <https://www.youtube.com/watch?v=vKcA094FSMk>

Demo: Gemma 2 architecture: JAX, Flax, and more

<sup>31</sup> <https://deepmind.google/discover/blog/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology/>  
AlphaFold: a solution to a 50-year-old grand challenge in biology

<sup>32</sup> <https://cloud.google.com/blog/products/ai-machine-learning/guide-to-jax-for-pytorch-developers>  
The PyTorch developer's guide to JAX fundamentals

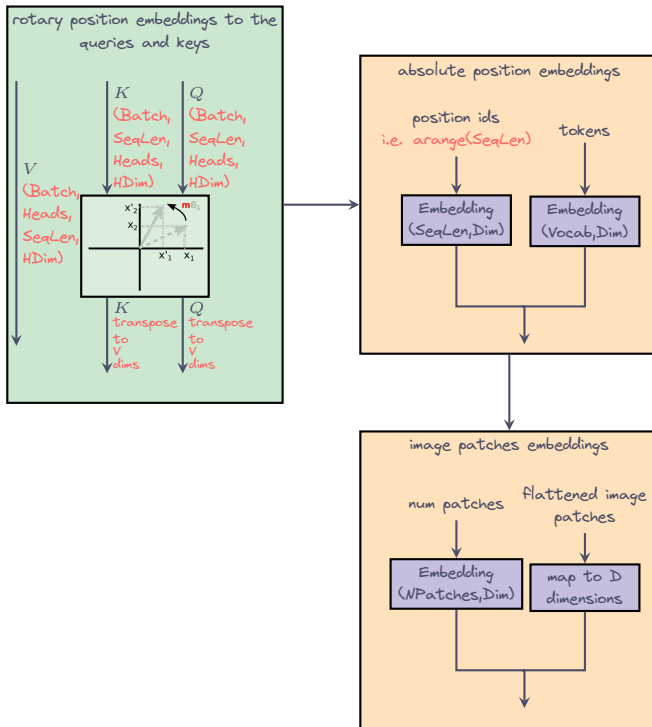
<sup>33</sup> [arXiv:2010.11929](https://arxiv.org/abs/2010.11929)  
An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale  
Dosovitskiy et al. 2020

<sup>34</sup> [ViT PyTorch vs JAX training benchmarks on Vertex AI Training Platform](#)

<sup>35</sup> <https://www.youtube.com/watch?v=eJfJRyXEHZ0>  
Announcing Gemma 3n Preview: Powerful, Efficient, Mobile-First AI

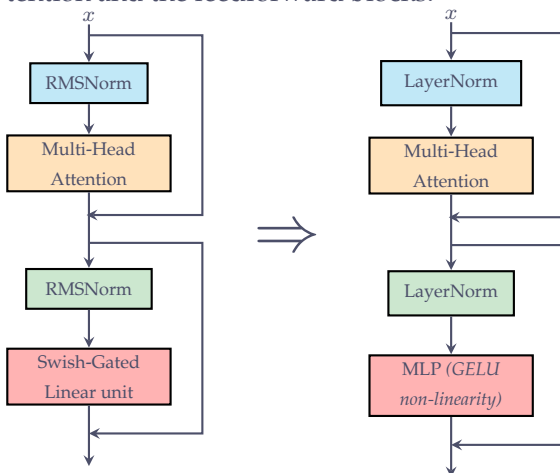
## Dumbing down Llama2 to base ViT

First looking at the inner most part of Llama2, the Scaled Dot Product Attention, the rotary positional embedding to the queries and keys is removed and instead the GPT-2 absolute positional embeddings used, however in this case the image patches are considered instead of tokens.



Moreover, ViT has no masking of the intermediate attention scores hence its attention mechanism is non-causal.

Within the Transformer Block, normalization changes from *RMSNorm* to *LayerNorm*, computed for the inputs before the attention and the feedforward blocks.



## Understanding the image embeddings for ViT

Going onto the interesting part of what makes this pioneering research awesome, is how the image patches is handled. As described in [page 3 section 3.1 of the paper<sup>33</sup>](#), we reshape the image  $x \in R^{H \times W \times C}$  into a sequence of flattened 2D patches  $x_p \in R^{N \times (P^2 \cdot C)}$ , where  $(H, W)$  is the resolution of the original image,  $C$  is the number of channels,  $(P, P)$  is the resolution of each image patch, and  $N = HW / P^2$  is the resulting number of patches, which also serves as the effective input sequence length for the Transformer.

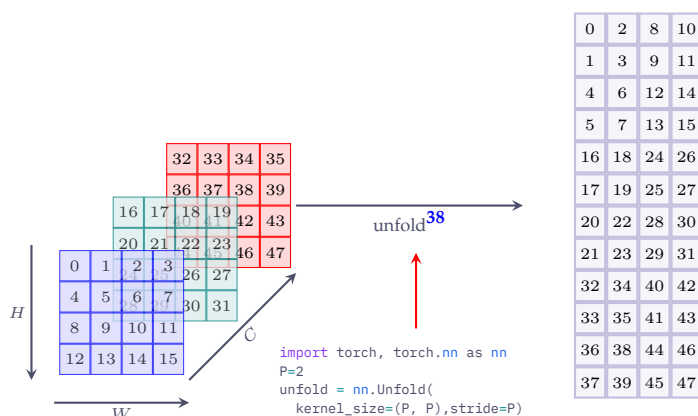
Let's implement this process in two ways

- using torch unfold
- using einops Rearrange

### torch unfold

PyTorch is an awesome framework, and with it comes one nice functionality, *torch unfold*<sup>36</sup>, the functionality behind the very core convolution operations in convolutional neural networks. So how does it work?

Imagine we have a 3-channel tensor, then *unfold* extracts the patch of a given kernel size across all channels in the tensor and unrolls it to a single column, before proceeding to the next patch as noted by the stride<sup>37</sup>.



Next, let's create patches from an image and visualize them for the different kernel sizes. Let's consider this image I once used in the *From Tensors to Residual Learning*<sup>39</sup> course which I taught in PyTorch and focused on the fundamentals of Deep Learning from the mathematics of Calculus to implementing the different variants of residual network architectures.

<sup>36</sup>[Unfold](#)

PyTorch API reference

<sup>37</sup>the tensor needs extrusion to the batch dimension since unfold only supports 4-D tensors

<sup>38</sup>Note that *unfold* gives us the flattened 2D patches as a transpose of the expected 2D patches in the paper, since we have in the resulting visual  $(P^2 \cdot C) \times N$  when we need its transpose, denoted as  $x_p$  in the paper of dims  $N \times (P^2 \cdot C)$ . This will be handy later on!

<sup>39</sup>[From Tensors To Residual Learning](#)

From the math of tensors to the implementation of residual learning

Huggingface datasets 🐶 come in handy for this, getting the image which is a *Pillow* instance which we then transform to a *torch Tensor*

```
from datasets import load_dataset
from torchvision.transforms.v2.functional import (
    pil_to_tensor, resize)
# function to resize the image to 240 by 240
resize_fn = lambda x, size=240: resize(
    x, size=[size,size]).to(torch.float32)

dataset = load_dataset("huggingface/cats-image")
image = dataset['test']['image'][0]
image = pil_to_tensor(image)
image = resize_fn(image)
C,H,W=image.shape
```

Let's of course initialize the unfold instance. We want to get, for now, 4 patches from the image, this means the patch size is

$$N = HW/P^2 \Rightarrow 4 = (240/P)^2 \Rightarrow P = 120$$

```
P=120
unfold = nn.Unfold(kernel_size=(P, P),stride=P)
patches = unfold(image)
print(patches.shape) # torch.Size([1, 43200, 4])
```

and so the patches can be converted to images as

```
def extractPatch(patches, index, P):
    patch = patches[...,index].view(1,3,P,P)
    patch = patch.squeeze(0).permute(1,2,0)
    return patch.to(torch.uint8).numpy()
```

and visualize<sup>40</sup> the image patches using the code

```
showPatches(patches,N,P)
```

to get



40

```
import matplotlib.pyplot as plt

def showPatches(patches, N, P):
    rows=cols=int(N**.5)
    for i in range(N):
        plt.subplot(rows,cols,i+1)
        patch_image = extractPatch(patches, i, P)
        plt.imshow(
            patch_image,aspect = "auto"
        )
        plt.tight_layout()
        plt.xticks([]); plt.yticks([])
    plt.subplots_adjust(
        hspace=0.1,wspace=0.1 # wspace => aspect auto
    )
    plt.show()
```

## The next big thing, einops<sup>41</sup>

This library greatly simplifies a lot of incredible operations, and is greatly used even in simplifying high dimensional matrix-multiplication heavy layers in many model graphs. The docs is wonderful and on the side notes for in-depth review. In this case, we want to get the image dimensions, and then get the height and width as integer multiples of the patch size, but we need to be cautious on how we handle the channel dimensions, that is, how we translate the unfold operation as either channel dimensions first or spatial dimensions first. This change in dimensions is implemented using einops as

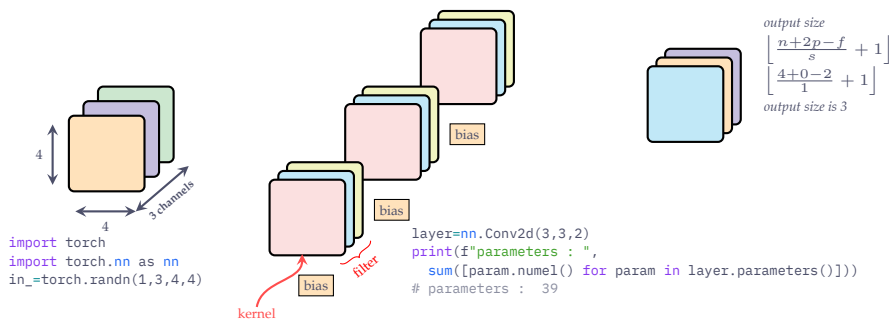
```
from einops.layers.torch import Rearrange
rearr = Rearrange(
    'b c (h p1) (w p2) -> b (c p1 p2) (h w)',
    p1 = P, p2 = P)
```

considering that the image is already a 4D tensor. The above operations says, hey, let's take subsets of the spatial dimensions across all channels, then rearrange the dimensions to a batch of these tensors of size  $P^2 \cdot C$ ,  $N$  of such, dimensions denoted as  $(1, P^2 \cdot C, N)$ . Reminder that this is the transpose of  $x_p$ . It can be invoked to get the image patches which then gives the same results as the visual just above.<sup>42</sup>

```
patches_again = rearr(image)
showPatches(patches_again, N, P)
```

## torch convolution

I've always loved convolution, because I love signal processing, and I love image processing, and I love Computer Vision. This operation is core in many topics across these domains. Convolution neural networks is inspired by the biological cortex and I've gone through it in-depth in my course<sup>43</sup>.



<sup>41</sup><https://einops.rocks/pytorch-examples.html>  
Writing a better code with pytorch and einops

<sup>42</sup>Note that as we understood from <sup>38</sup>, we have to transpose how we actually got the patches to

```
Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)',
          p1=P, p2=P)
```

which then gives us the 2D flattened patches  
 $x_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$ .

<sup>43</sup><https://youcanjustbuild.com/courses/from-tensors-to-residual-learning/foundations>  
The foundations of residual learning



### Now onto implementing the image embeddings

Using convolution, we implicitly get the image patches and linearly project to D dimensions directly. The kernel size becomes the size of the needed patch with the stride being the same, the output channels size become  $D^{44}$ . This layer is also essential given it is a trainable layer, as opposed to the two previously looked-into ops that weren't.

```
patch_layer=nn.Conv2d(3, 768, kernel_size=16, stride=16)
```

Based on the output size from the convolution formula, we get

$$\left\lfloor \frac{256 - 16}{16} + 1 \right\rfloor = 16$$

which begets the output size of  $16 \times 16$  with 768 channels in the order (1, 768, 16, 16). Looking into this output, it aligns more with  $(1, D, \sqrt{N}, \sqrt{N})$  when what we need is  $(1, N, D)$ . Hence, we consider two operations, combining the last two dimensions then transposing the now new last two dimensions, programmatically implemented as

```
in_ = torch.randn(1,3,256,256)
out = patch_layer(in_)
out = out.flatten(2) # flatten from dim 2, zero-indexed
patch_embs = out.transpose(1,2)
```

The output of this projection is known as the *patch embeddings*. We then need to prepend a class token<sup>45</sup> to the sequence of embedded patches

$$[x_{\text{class}}; x_p^1 \mathbf{E}; x_p^2 \mathbf{E}; \dots; x_p^N \mathbf{E}]$$

```
cls_token = nn.Parameter(torch.randn(1, 1, dim))
x = torch.cat((cls_token, patch_embs), dim=1)
```

### Adding positional embeddings to the patch embeddings

By simplifying their learning embedding from the advanced 2D-aware position embedding due to no significant performance, the paper uses 1D position embeddings defined by  $E_{pos} \in R^{(N+1) \times D}$ .

```
position_embeddings = nn.Parameter(
    torch.randn(1, N+1, dim))
```

which is then added to the augmented patch embeddings

<sup>44</sup>the original ViT uses image resized to 256 with patch size of 16 and D being 768 for ViT-Base 16. *patch\_dim* is basically  $C \times P^2$ .

<sup>45</sup>In the interesting implementing of *Vision Transformers*, some 10x engineers opted for toggling between mean pooling and cls tokens computed with the original patch embeddings. You can see their awesome implementation of such in their codebase [https://github.com/lucidrains/vit-pytorch/blob/main/vit\\_pytorch/vit.py](https://github.com/lucidrains/vit-pytorch/blob/main/vit_pytorch/vit.py)

```
x += pos_embeddings[:,(N+1)]
```

and that is it to give us the whole implementation as

```
class ImageEmbeddings(nn.Module):
    def __init__(self, H, W, P, dim):
        super(ImageEmbeddings,self).__init__()
        N = int((H*W)/(P**2)); self.N = N
        assert H%P==0 and W%P==0, \
            "image size must be integer multiple of patch"
        self.conv_then_project = nn.Conv2d(
            3,out_channels=dim,kernel_size=P,stride=P)
        self.cls_toks = nn.Parameter(torch.randn(1, 1,
            dim))
        self.pos_embeds =
            nn.Parameter(torch.randn(1,N+1,dim))
    def forward(self,image):
        x = self.conv_then_project(image)
        x = x.flatten(2); x = x.transpose(1,2)
        cls_toks = self.cls_toks.expand(x.shape[0],-1,-1)
        x = torch.cat((cls_toks,x),dim=1)
        x += self.pos_embeds[:,:(self.N+1)]
        return x
```

### The Looping transformer Block

From the embeddings layer, ViT implements a loop of alternating MultiHeadAttention and MLP blocks, shown previously in the *RMSNorm* to *LayerNorm* section, with LayerNorm applied before every block, and residual connection after every block<sup>46</sup>. As also discussed, the MultiHeadAttention is not masked since it is bidirectional. The Hence, the resulting implementation for the MultiHeadAttention removes RoPe mechanism and masking to be

```
class MultiHeadAttention(nn.Module):
    def __init__(self,d_in,d_out,n_heads,bias=True):
        super(MultiHeadAttention,self).__init__()
        assert d_out%n_heads==0, "d_out must be integer
            multiple of n_heads"
        self.head_dim = int(d_out / n_heads)
        self.n_heads = n_heads
        self.Wq = nn.Linear(d_in,d_out,bias=bias)
        self.Wk = nn.Linear(d_in,d_out,bias=bias)
        self.Wv = nn.Linear(d_in,d_out,bias=bias)
        self.Wo = nn.Linear(d_out,d_out,bias=bias)
```

<sup>46</sup> ViT-Base has the two blocks, MultiHeadAttention and MLP, looped 12 times, meaning 12 layers of the Transformer Block. ViT-Base encompasses ViT-B/16 and ViT-B/32, with the values after the slash being for the patch sizes.

```

# ...
def forward(self, x):
    B, seq_len, d_in = x.shape
    q = self.Wq(x); k = self.Wk(x); v = self.Wv(x)
    q = q.view(B, seq_len, self.n_heads, self.head_dim)
    k = k.view(B, seq_len, self.n_heads, self.head_dim)
    v = v.view(B, seq_len, self.n_heads, self.head_dim)
    q = q.transpose(1, 2)
    k = k.transpose(1, 2)
    v = v.transpose(1, 2)
    scores = q @ k.transpose(-1, -2)
    scores = scores / k.shape[-1]**.5
    norm_scores =
        nn.functional.softmax(scores, dim=-1)
    y = norm_scores @ v
    out = y.transpose(1, 2).contiguous().view(
        B, seq_len, d_in)
    out = self.Wo(out)
    return out

```

The next block after the residual connection proceeding the MHA is two linear transformations separated by a GELU activation.

```

class MLP(nn.Module):
    def __init__(self, dim, hidden_dim, bias=True):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(dim, hidden_dim, bias=bias)
        self.fc2 = nn.Linear(hidden_dim, dim, bias=bias)
        self.act = nn.GELU(approximate='tanh')
    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(self.act(x))
        return x

```

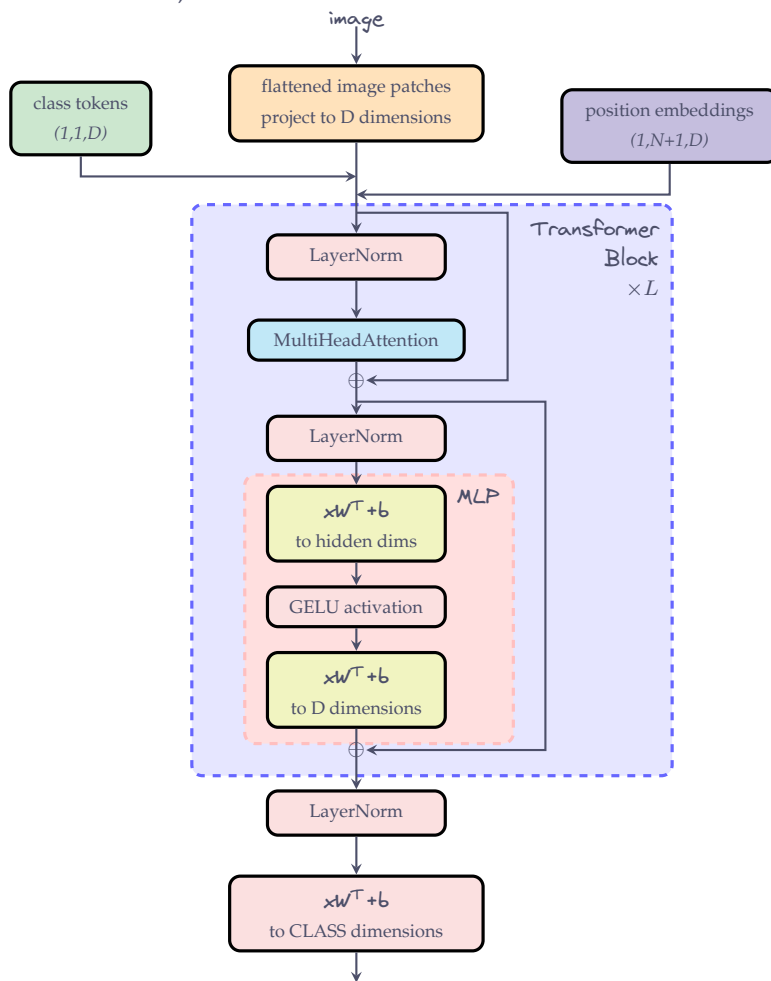
And hence the complete Transformer Block with LayerNorm before and residual connection after the MHA and the MLP<sup>47</sup> is beautifully implemented as a PyTorch module as follows

```

class ViTBlock(nn.Module):
    def __init__(self, CONFIG):
        super(ViTBlock, self).__init__()
        self.ln1 = nn.LayerNorm(CONFIG.D_IN)
        self.attn =
            MultiHeadAttention(CONFIG.D_IN, CONFIG.D_OUT, CONFIG.HEADS)
        self.ln2 = nn.LayerNorm(CONFIG.D_IN)
        self.mlp = MLP(CONFIG.D_IN, CONFIG.HIDDEN_DIM)
    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x

```

And with that, the whole model stands out as



```
class VisionTransformer(nn.Module):
    def __init__(self, CONFIG):
        super(VisionTransformer, self).__init__()
        self.image_embeddings = ImageEmbeddings(
            CONFIG.H, CONFIG.W,
            CONFIG.P, CONFIG.D_IN
        )
        self.vit_blocks = nn.Sequential(
            *[ViTBlock(CONFIG)
              for _ in range(CONFIG.LAYERS)])
        self.norm = nn.LayerNorm(CONFIG.D_IN)
        self.out_linear =
            nn.Linear(CONFIG.D_IN, CONFIG.CLASSES)
    def forward(self, x):
        x = self.image_embeddings(x)
        x = self.vit_blocks(x)
        x = self.norm(x[:, 0])
        x = self.out_linear(x)
        return x
```

## Inferencing using ViT-B/16

The specific model *ViT-B/16* is named so because it uses a patch size of 16 and is the base variant of ViT from those implemented in the paper, all base variants having features

Layers	Hidden size $D$	MLP size	Heads	Params
12	768	3072	12	86M

Taking a model roughly pretrained on the cifar10 dataset, let's copy the weights to our model and do some rough inferencing before we then explore the following variants that we'll now optimize, *ViT-L16*, *ViT-H14*, *ViT-g14*, *ViT-G14*, with the last two variants presented in the paper<sup>48</sup> that looks into the scaling properties as a key to designing future generations effectively. The inference pipeline is interestingly implemented and then can be invoked as<sup>49</sup>

```
vit = VisionTransformer(CONFIG)
weights = torch.load(
    "../models/vit_b_16_pretrained_cifar10.pth",
    weights_only=True, map_location='cpu')
copy_model_weights(vit, weights)
vit = vit.to(device); vit.eval()

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
inv_transforms = Compose([
    Normalize(mean = [-i/j for i,j in zip(MEAN,STD)],
              std = [1/i for i in STD]),
    Lambda(lambda x: x * 255),
    Lambda(lambda x: x.permute(1,2,0)),
    Lambda(lambda x: x.to(torch.uint8))
])

iter_loader = iter(test_loader)
images, labels = next(iter_loader)
images = images.to(device)
with torch.no_grad():
    preds = vit(images)

for i, (image, label) in enumerate(zip(images, labels)):
    plt.subplot(4,4,i+1)
    pred = preds[i,:].argmax(dim=-1).item()
    cls = classes[pred]
    image = image.cpu().squeeze()
    inv_image = inv_transforms(image)
    plt.imshow(inv_image, aspect='auto')
    plt.axis('off')
    plt.title(f'{cls}', color='g' if pred == label else 'r')
plt.subplots_adjust(hspace=0.5, wspace=0.5)
plt.show()
```

<sup>48</sup> [arXiv:2106.04560](https://arxiv.org/abs/2106.04560)

Scaling Vision Transformers

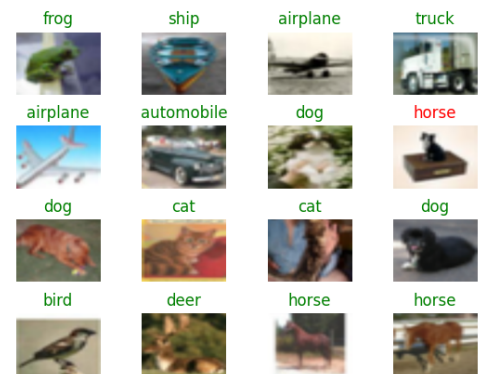
Zhai et al. 2022 (The infamous Lucas Beyer is also a co-author on this)

<sup>49</sup> the core\_utils.py in the directory

<https://github.com/Marvin-desmond/ScalingViTsAcrossTrainingCompute/blob/main/transformerViT> implements the weight copying function and the rest of the inference code can be found in the core.py file with the configs for ViT-B/16 being

```
class CONFIG:
    P = 16
    H = 224
    W = 224
    D_IN = 768
    D_OUT = D_IN
    HEADS = 12
    LAYERS = 12
    HIDDEN_DIM = D_IN * 4
    CLASSES = 10
```

the result of the inference gives



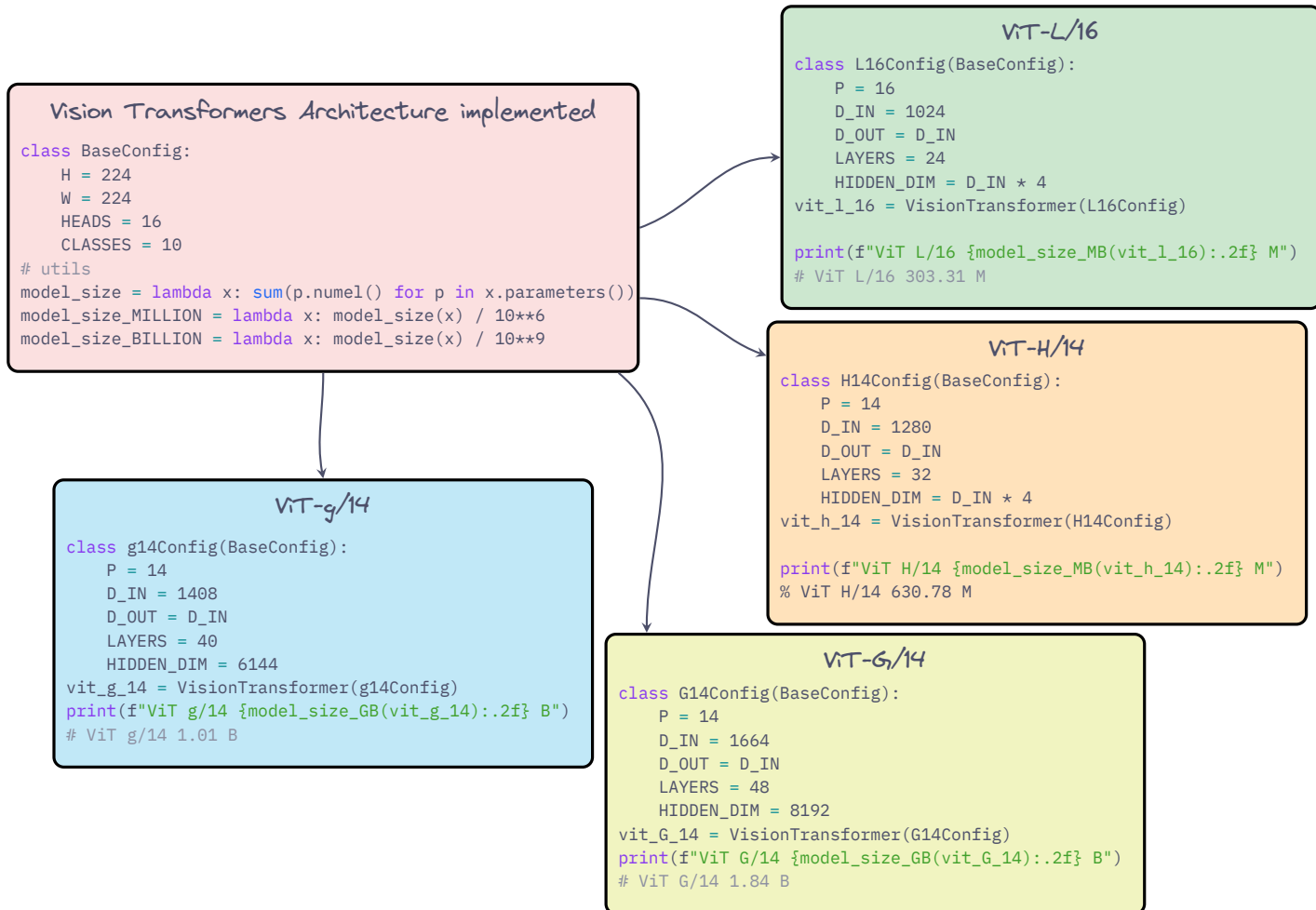
## The Four Variants of the "Optimization Apocalypse"

Revisiting the original *Vision Transformers* paper, we see that ViT-L16 can be grouped as a large variant with patch size 16, and ViT-H14 can be put in the huge variant with patch size 14. Image size is 224 pixels for the experiments done in the paper hence we'll hold on that for the rest of the article section.

The variants ViT-g14 and ViT-G14 are from the next paper <sup>48</sup> and are scaled versions of the earlier models trained to benchmark performance during scaling under different conditions. The summary for the models are then thus (as extracted from Table 1 and 2 of the respective papers)

Type/Patch	Depth (Layers)	Dimensions $D$	Intermediate MLP Size	Heads	Params
L/16	24	1024	4096	16	307M
H/14	32	1280	5120	16	632M
g/14	40	1408	6144	16	1.011B
G/14	48	1664	8192	16	1.843B

which then translates programmatically to



### *The naive data pipeline*

The benchmarking of the ViT variants by Google DeepMind from the naive pipeline then scales it to data parallel pipeline using PyTorch's *Distributed Data Parallel* for models that fit into a single gpu,( refer to the github doc<sup>34</sup>). We may not explore huge models that need be fit in many gpus given the article is already too long enough, but this will be interestingly explored sooner in a later article that will also go into the interesting DiLoCo<sup>50</sup> which is used to revolutionize multi-node training in a high fault networked training<sup>51</sup>. For the ViT-G14, the Fully Sharded Data Parallel (FSDP) is used instead.

Let's now start on building the naive pipeline and use the time in hours per epoch for how much speed improvement we get during optimization even in the model graph using *torch.compile* and float precision reduction in the computations.

### *Everything starts with cifar10<sup>52</sup> dataset*

This dataset is infamous for scaling of models, and awesome papers such as one by Keller Jordan<sup>53</sup>, shows how much still can be explored based on it. It can be loaded from the torchvision datasets

```
import torch, torchvision
from torchvision.transforms.v2 import (
    Compose, ToImage, Resize, ToDtype, Normalize)
SEED = 42; torch.manual_seed(SEED)
MEAN = [0.485,0.456,0.406]; STD = [0.229,0.224,0.225]
transforms = Compose([
    Resize((224, 224)),ToImage(),
    ToDtype(torch.float32,scale=True),Normalize(MEAN,STD),
])

train_data = torchvision.datasets.CIFAR10(
    root=../datasets, train=True, transform=transforms,
    download=True)
test_data = torchvision.datasets.CIFAR10(
    root=../datasets, train=False, transform=transforms,
    download=True)

print(f"Train images: ", len(train_data)) # Train images: 50000
print(f"Test images: ", len(test_data)) # Test images: 10000
```

The batch will be scaled as per the training limits as from the table in the github optimization docs, but for now, let's consider Yann Lecun's

*Friends don't let friends use mini-batches larger than 32*

<sup>50</sup> [arXiv:2311.08105](#)

DiLoCo: Distributed Low-Communication Training of Language Models

Douillard et al. 2023

<sup>51</sup> [Fault Tolerant Llama: training with 2000 synthetic failures every 15 seconds and no checkpoints on Crusoe L40S](#)

Tristan Rice, Howard Huang

June 20, 2025

<sup>52</sup> <https://www.cs.toronto.edu/~kriz/cifar.html>

The CIFAR-10 dataset

Thanks Hinton and the team

<sup>53</sup> [arXiv:2404.00498](#)

94% on CIFAR-10 in 3.29 Seconds on a Single GPU

Keller Jordan

Now that we know the batch size to use, let's implement the iterable for batched data processing

```
from torch.utils.data import DataLoader
BATCH=32
train_loader = DataLoader(train_data, batch_size=BATCH, shuffle=True)
test_loader = DataLoader(test_data, batch_size=BATCH, shuffle=True)
```

And then now, let's start with the model benchmark for ViT-L/16. Instantiating it becomes easy, considering we've gone through it clearly.

```
vit_l_16 = VisionTransformer(L16Config)
device = (torch.accelerator.current_accelerator().type
          if torch.accelerator.is_available() else "cpu")
vit_l_16 = vit_l_16.to(device)
vit_l_16.train()
```

Next, let's define the optimizer that updates the weights with respect to the gradients computed and the cross entropy<sup>54</sup> loss function, the one for which invoking backward propagation computes the needed gradients. I explain a lot more about gradient computation and backpropagation in this course<sup>55</sup> and specifically the section for which the link directs.

```
optimizer = torch.optim.SGD(
    vit_l_16.parameters(), lr=1e-3, momentum=.9)
criterion = torch.nn.CrossEntropyLoss()
```

onto training for a single epoch and computing the time per epoch hours, then

```
import time
from tqdm import tqdm
start = time.time()
running_loss = 0.0
for i, (images, labels) in enumerate(tqdm(train_loader)):
    images = images.to(device)
    actual = labels.to(device)
    optimizer.zero_grad()
    predicted = vit_l_16(images)
    loss = criterion(predicted, actual)
    running_loss += loss.item()
    if i % 200 == 0:
        last_loss = running_loss / 200
        print(' BATCH {} LOSS: {:.2f}'.format(i + 1, last_loss))
        running_loss = 0.
    loss.backward()
    optimizer.step()
stop = time.time()
print(f"time : {(stop - start) / 3600:.5f} hours per epoch")
```

<sup>54</sup>This video awesomely delves into probability and cross-entropy. Its baseline concept is entropy, which is a measure of uncertainty.



The Key Equation Behind Probability  
by Artem Kirsanov

<sup>55</sup>From Tensors to Residual Learning  
Playing with Grads



### *Time in Hours per Epoch: Part One*

The metric of priority is time in hours per epoch. Considering batch size of 32 for a start, trying to run this on the *T4* free GPU tier of 16GB VRAM in Google Colab crashes since training requires more memory than the available VRAM, so, the options to not run out of memory involve:

- reduce the batch size, training will run but will be approximately twice the time, and we are not trying to increase the time, we need to reduce it.
- mixed-precision training, trains some layers, i.e. convolution layers on float16 which makes them much faster with the rest like reduction ops on float32 <sup>56</sup>.
- quantization-aware training, involves clamping and rounding floating-point values during training to simulate int8 precision <sup>57</sup>.
- using LoRA (Low-Rank Adaptation) and its variants which uses lower rank matrices of the matrix multiplication layers i.e. linear layers <sup>58</sup>.

Of the common training techniques mentioned, we'll instead focus on Data Distributed Parallel, Fully Sharded Data Parallel, reduced precision training, and optimizing the model graph using Jax. Summarily, we'll focus on scalable multi-GPU and TPU training. The compute for these techniques will be remotely spawned with the much needed GPU RAM and tunneled via ssh.

### *Remote Compute*

For this section, the platforms *RunPod* and *Vast* are to be used for spawning the compute needed, you can choose either of these for your workloads, but now for the connection, ssh is the just needed tool for connecting and running commands and programs in the compute.

### *What is Secure Shell (SSH)?*

This is a secure protocol that enables logging into remote compute, running commands, copying files to and from the compute, and installing tools. Think of it as a way to always use your normal computer via the terminal and not by using Graphical Interfaces <sup>59</sup>.

<sup>56</sup> [Automatic Mixed Precision](#)

*PyTorch blog by Michael Carilli, Created 2020, Updated 2025*

<sup>57</sup> [Quantization](#)

*Quantization Aware Training for Static Quantization*

<sup>58</sup> [Fine-Tuning Llama2 with LoRA](#)

<sup>59</sup> Well, you could use graphical interfaces, but by port forwarding, and that is beyond the scope of this article.

## Spawn a GPU Compute in RunPod

The cited article<sup>60</sup> goes into generating an ssh key, setting up the public key in the platform, then ssh-ing into the spawned remote compute. I'd recommend two great tools that will help navigate between your local and remote compute with great flexibility

- helix editor<sup>61</sup>, an terminal-based text editor that does not need customizations by having to write configs in a complex functional language. Neat for and editing programs and writing automation tools within the remote compute.
- tmux<sup>62</sup>, a tool that helps in managing multiple sessions, local and remote, without having to switch terminals or apps. Great for terminal-based multi-tasking and persistent ssh tunneling.

Going into runpod, we need the *A40, 48GB VRAM* pod. This allows for inference given no parallelism. Next, as per specified in the benchmark of reference<sup>63</sup>, we'll upgrade to a compute of approximately *40GB* per gpu, 8 counts of such.

Runpod gives us this compute for \$0.4/hour, a nice option considering other options like [Vast.ai](#) gives us the same for \$0.471/hour, not much difference for small training, but considering days of training, becomes a significant difference. After moving all necessary files<sup>64</sup> to the spawned remote compute using the command

```
scp -P $PORT {core_utils,train_official}.py
root@$IP:/root/
```

The time per epoch hours attained is

```
time : 0.33440 hours per epoch
```

or to say it another way, 33 minutes. However, one thing is to be noted from ViT benchmarks for the *ViT-L/16*, the global batch size is 512, meaning for the 8 optimal gpus used, the batch size per gpu then becomes 64, so, running another epoch for the now modified batch size of 64 gives the time per epoch of

```
time : 0.32292 hours per epoch
```

The next section then goes into the start of parallelism and how to beat the time achieved by Google (or approximate it) of

```
time : 0.00722251 hours per epoch
```

<sup>60</sup>Use SSH

<sup>61</sup>Helix Editor Official Site

<sup>62</sup>

[Tmux Getting Started](#)

<sup>63</sup>Benchmark Table

Model	Framework	Accelerator	Global batch-size	Precision	Img/Sec	Time in hrs per epoch (50k images)	Per hour cost (USD) for 1 epoch on Vertex AI	Cost (USD) per epoch on Vertex AI
ViT-L/16 (300M)	PyTorch Multi GPU with DDP strategy	8 A100-40GB (1 host)	512	float16	1823	0.00722251	34.482189	0.249
	JAX TPU	TPU v3 8 cores	512	bfloat16	730	0.01851851	10.12	0.1874
	JAX TPU POD	TPU v3 32 cores (4 hosts)	2048	bfloat16	3320	0.00459896	36.8	0.1692
	JAX GPU	8 A100-40GB (1 host)	128	bfloat16	395	0.04692192	4.310274	0.2022
ViT-H/14 (600M)	PyTorch Multi GPU with DDP strategy	8 A100-40GB (1 host)	128	float16	652	0.02516103	34.482189	0.8676
	JAX TPU POD	TPU v3 32 cores (4 hosts)	1024	bfloat16	695	0.01408609	36.8	0.5163
	JAX Multi GPU	8 A100-40GB (1 host)	256	bfloat16	781	0.01778346	34.482189	0.6132
	PyTorch Multi GPU with DDP strategy	8 A100-40GB (1 host)	128	float16	608	0.03404139	34.482189	1.1738
ViT-g/14 (1B)	PyTorch Multi GPU with DDP strategy	8 A100-40GB (1 host)	128	float16	508	0.02388771	36.8	0.8827
	JAX TPU POD	TPU v3 32 cores (4 hosts)	512	bfloat16	579	0.02025967	34.482189	1.1215
	JAX Multi GPU	8 A100-40GB (1 host)	128	bfloat16	527			
	PyTorch Multi GPU with DeepSpeed ZeRO (Stage3) strategy	8 A100-40GB (1 host)	256	float16	395	0.07122507	34.482189	2.4559
ViT-g/14 (1.8B)	JAX TPU POD with FSDP	TPU v3 32 cores (4 hosts)	4096	bfloat16	333	0.04170837	36.8	1.5346
	JAX Multi GPU with FSDP	8 A100-40GB (1 host)	256	bfloat16	395	0.04709097	34.482189	1.6234

<sup>64</sup>the files can be found in the repository and the folder <https://github.com/Marvin-desmond/ScalingViTsAcrossTrainingCompute/blob/main/transformerViT/>

## Message Passing Interface (MPI)

As adapted from the book, Programming Massively Parallel Processors<sup>65</sup>, a MPI is

*a programming interface in which computing nodes in a cluster do not share memory. This memory isolation strategy hence dictates that all data sharing and interaction within must be done through explicit message passing, as common in High Performance Computing(HPC). In HPC, apps written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Due to this high number of nodes and lack of shared memory, the amount of effort then that is needed to port an application into MPI can be quite high across computing nodes.*

This tool is commonly implemented in C, with its Python ported version<sup>66</sup> to be used by high-level compute developers. PyTorch implements all this in Data Distributed Parallel which has been tuned for multi-GPU hence is much much faster, one that is part of this article that is coming after this. However, using MPI natively gives more control.

After installing your version of MPI<sup>67</sup>, then compiling it will be by using *mpicc* instead of *gcc*. This *hello world* code of MPI in C

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int name_len, world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(processor_name, &name_len);
    char *s_or_p_processor = world_size < 2 ? "processor" :
        "processors";
    printf("Hello world from processor %s out of %d %s\n",
        processor_name, world_size, s_or_p_processor);
    MPI_Finalize();
}
```

thus when compiled gives

```
> mpicc -o hello_mpi hello_mpi.c && ./hello_mpi
Hello world from processor Marvins-MacBook-Pro-4.local
out of 1 processor
```

<sup>65</sup>Programming Massively Parallel Processors  
A Hands-on Approach  
4th Edition - May 28, 2022  
[Buy Link](#)

<sup>66</sup>[mpi4py](#)

<sup>67</sup><https://www.open-mpi.org/software/ompi/v5.0/>

So knowing that MPI uses multiprocessing, multiple processes are spawned, each of these processes having their own memory. This then provides parallelism as these processes are independent of each other and do not have to wait for one another, executing across multiple input values as the input data is distributed across processes.

### Data Parallelism

From the context of model training, considering you have a very large dataset, then data parallelism means that you take your model and copy it across multiple machines (GPUs), then different non-overlapping batches of data are uniquely sent across the different machines, forward pass done in each machine, gradients locally computed, then the computed gradients are then aggregated across all processes, then averaged on a master node by the number of gpus to to give global gradients, before the processes are sent the new global gradients that then are used to update the models on each of those machines terms of their parameters. This ensures that all models are thus a copy of each other.

From this, then two collective communication strategies are key

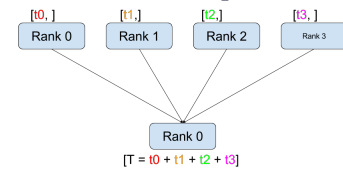
- reduce (or all-reduce) <sup>68</sup>
- broadcast <sup>69</sup>

Let's consider an array with 4 values representative of four batches, and so we send the values to the four processes <sup>70</sup>.

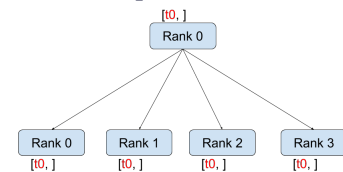
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, world_size, scattered_val;
    int values[4] = {39, 72, 129, 42};
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Scatter(&values, 1, MPI_INT, &scattered_val, 1,
        MPI_INT, 0, MPI_COMM_WORLD);
    printf("Processor %d received %d\n", rank,
        scattered_val);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

<sup>68</sup>reduce ~ computes a reduction operation then stores the result in a master node, all-reduce instead stores the result in all the processes



<sup>69</sup>broadcast ~ copies the computed global gradient from the master process to the other processes.



diagrams courtesy of

[Writing Distributed Applications with PyTorch](#)

<sup>70</sup>and run the program as

```
> mpicc -o hello_mpi hello_mpi.c
> mpirun -np 4 ./hello_mpi
Processor 0 received 39
Processor 1 received 72
Processor 2 received 129
Processor 3 received 42
```

the MPI barrier is since the forward and backward pass for each process needs to finish and all local gradients sent to the master process, so it synchronizes all processes upto when the last process sends the local gradient to the master node.

### Simple Reduce and Broadcast

Right before the line `MPI_Finalize()`, consider that as our starting point. Assuming each process involves computation that simulates the gpu computation of the forward and backward pass, which we then gather the results to a master process, then the additional code is <sup>71</sup>

```
// ...
//a computation to simulate a forward and back pass
scattered_val = scattered_val * 0.5;
printf("Processor %d updated value %d\n",rank,
       scattered_val);
MPI_Barrier(MPI_COMM_WORLD);
// gather to rank zero
int *rbuf;
if (rank == 0) {
    rbuf = (int *)malloc(world_size*sizeof(int));
}
MPI_Gather(&scattered_val, 1, MPI_INT, rbuf, 1,
          MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0){
    for (int i=0; i < world_size; i++) {
        printf("Rank %d: rbuf[%d] = %d\n", rank, i,
              rbuf[i]);
    }
    free(rbuf);
}
//...
```

However, we need not to do a gather, but a reduce, so then doing a reduce with MPI leads to the above code being modified, from the line `int *rbuf`, as

```
// ...
// gather to rank zero
int global_avg;
MPI_Reduce(&scattered_val, &global_avg, 1, MPI_INT,
          MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0){
    printf("Rank %d: sum %d average %.1f\n",
          rank, global_avg, (float)global_avg/world_size);
}
// ...
```

```
Rank 0: sum 140 average 35.0
```

<sup>71</sup> the result of the program is thus

```
> mpicc -o hello_mpi hello_mpi.c
> mpirun -np 4 ./hello_mpi
Processor 0 received 39
Processor 0 updated value 19
Processor 1 received 72
Processor 1 updated value 36
Processor 2 received 129
Processor 2 updated value 64
Processor 3 received 42
Processor 3 updated value 21
Rank 0: rbuf[0] = 19
Rank 0: rbuf[1] = 36
Rank 0: rbuf[2] = 64
Rank 0: rbuf[3] = 21
```

Now let's broadcast the resulting average value, signifying the computed global gradients, to all the processes (nodes)<sup>72</sup>.

```
// ...
// Broadcast the result to all the processes
float broadcast_val = 0.0f;
if (rank == 0){
    broadcast_val = (float)global_avg/world_size;
}
MPI_Bcast(&broadcast_val, 1, MPI_FLOAT, 0,
        MPI_COMM_WORLD);
printf("Processor %d final value %f\n",rank,
        broadcast_val);
// ...
```

We actually don't need *MPI\_Barrier* before the collective operations since they implicitly require synchronization before starting to execute.

### MPI in Python

Considering the first speedup of ViT training will use this library, *mpi4py*, let us rewrite the previous implementation of the reduce and broadcast in Python, with no need for explanations as it does the same operations<sup>73</sup>.

```
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
if rank == 0:
    data = [39, 72, 129, 42]
else:
    data = None
# simulate sending batches to nodes
data = comm.scatter(data, root=0)
print(f"Processor {rank} received {data}")
comm.Barrier()
# simulate forward and back pass
data = data * 0.5
print(f"Processor {rank} updated {data:.2f}")
# reduce
resulting_sum = np.array(0.0, dtype=np.float64)
comm.Reduce(
    np.array(data, dtype=np.float64),
    resulting_sum, op=MPI.SUM, root=0)
if rank == 0:
    print(f"Processor {rank} sum {resulting_sum:.2f}")
# broadcast the average
data = comm.bcast(resulting_sum/size, root=0)
print(f"Processor {rank} final {data:.2f}")
```

<sup>72</sup>the result of the program is

```
Processor 0 received 39
Processor 1 received 72
Processor 2 received 129
Processor 3 received 42
Processor 3 updated value 21
Processor 3 final value 35.000000
Processor 2 updated value 64
Processor 2 final value 35.000000
Processor 0 updated value 19
Rank 0: sum 140 average 35.0
Processor 0 final value 35.000000
Processor 1 updated value 36
Processor 1 final value 35.000000
```

<sup>73</sup>and the Python equivalent program using MPI is run as  
ensure *mpi4py* is installed using *pip install mpi4py*

```
mpiexec -n 4 python hello_parallel.py
```

which gives the expected result as

```
Processor 0 received 39
Processor 2 received 129
Processor 3 received 42
Processor 1 received 72
Processor 1 updated 36.00
Processor 3 updated 21.00
Processor 0 updated 19.50
Processor 2 updated 64.50
Processor 0 sum 141.00
Processor 0 final 35.25
Processor 2 final 35.25
Processor 1 final 35.25
Processor 3 final 35.25
```

### DDP, a message passing tool for multi-node GPUs

So, we are going to skip delving into mpi4py with torch and training (for now), because we need to build torch from scratch for MPI to work for GPUs, and the last time I tried to build a tool<sup>74</sup>, I did not like the patience I had to endure for it to complete. Distributed Data Parallel is a tool that does distributed computations for pipelines by assigning accelerators for the said computations and also facilitates the communication between different processes which can either be peer-to-peer communication or collective communications.

PyTorch, using the distributed package, creates a group of processes that are "device-aware" using the `torch.distributed.init_process_group`, hook them up with fast communication backends, `nccl`, `gloo` and `mpi`, then prepare your data pipeline and model implementation to work in this multi-process context. Based on this then, a few environment variables are used to glue everything together,

- **world\_size** ~ total number of GPUs in your cluster
- **rank** ~ the unique id of a GPU.
- **store** ~ a shared key-value store used for initializing the distributed process group by exchanging information like addresses and ports.
- **process group** ~ a subset of processes(GPUs) that communicate with each other. Default group includes all GPUs.

For CPU, backend to be used is `gloo` with the maximum world size being the available physical CPU cores, though it is not recommended to use all the cores as this may degrade performance. Making sure each process has the capability to communicate to each other can be done with

```
import os, sys, torch.distributed as dist
import torch.multiprocessing as mp
def init_process(rank, size, fn, backend="gloo"):
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(
        backend, rank=rank, world_size=size)
    fn()
    dist.barrier()
    dist.destroy_process_group()
```

## Spawning processes using `torch.multiprocessing`

With the communication between processes defined, spawning 4 processes is then done with <sup>75</sup>

```
def run():
    rank = dist.get_rank()
    size = dist.get_world_size()
    print(f"rank {rank} size {size}")
if __name__ == "__main__":
    world_size = 4 # os.cpu_count()
    mp.set_start_method("spawn")
    mp.spawn(fn=init_process, args=(world_size,run),
            nprocs=world_size, join=True)
```

Rewriting the previous implementations of collective communications in C and Python now in `torch.distributed`, the resulting code is<sup>76</sup>

```
import torch
def run(rank,size):
    tensor = torch.zeros(1)
    data = [39,72,129,42]
    tensor = torch.tensor([data[rank]],
        dtype=torch.float32)
    print(f"rank {rank} size {size} data {tensor}")
    dist.barrier() # for print convenience
    tensor = tensor * 0.5
    print(f"rank {rank} updated data to {tensor}")
    dist.barrier() # for print convenience
    # reduce op
    dist.reduce(tensor,0,dist.ReduceOp.SUM)
    if rank == 0:
        print(f"rank {rank} reduced value {tensor}")
    # broadcast the average
    if rank == 0:
        tensor = tensor / size
    dist.broadcast(tensor,src=0)
```

Note that *reduce* and *broadcast* can be reduced to *all\_reduce* as

```
dist.all_reduce(tensor,dist.ReduceOp.SUM)
tensor = tensor / size
print(f"rank {rank} final value {tensor}")
```

The reason for being redundant was to show how collective communication ops works for the different tools that are common.

<sup>75</sup>how spawn works is very interesting, it calls the function `init_process` as `init_process(i, *args)` where `i` is the process index/rank. This then means we don't need to pass the rank argument as needed by the `init_process`. Also, we can either pass rank and world\_size to the run function or use the `dist.get_rank()` and `dist.get_world_size()` for the variables, whichever works for you.

<sup>76</sup>output of Python code

```
rank 3 size 4 data tensor([42.])
rank 3 updated data to tensor([21.])
rank 3 final value tensor([35.2500])
rank 1 size 4 data tensor([72.])
rank 1 updated data to tensor([36.])
rank 1 final value tensor([35.2500])
rank 2 size 4 data tensor([129.])
rank 2 updated data to tensor([64.5000])
rank 2 final value tensor([35.2500])
rank 0 size 4 data tensor([39.])
rank 0 updated data to tensor([19.5000])
rank 0 reduced value tensor([141.])
rank 0 final value tensor([35.2500])
```



*Spawning processes using torch.multiprocessing*  
lorem ip