Scaling ViTs across Training Compute

by Marvin Mboya

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[1] 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[2] 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 hiddetn 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.

[1] arXiv:1706.03762
Attention is all you need
Vaswani et al. 2017

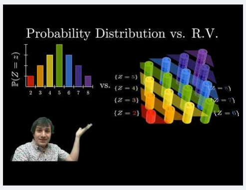
[2] RNNs can be understood using a special key word, **recurrent**, meaning to recur, where each hidden state would have a loop within itself and also includes the compounded outputs of all the previous hidden states, hugely based on the concept of a Markov model

Markov process is a stochastic process with the properties:

- number of posssible states is finite
- outcome at any state depends only on outcomes pf 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 [3].

Implementing a simple tokenizer based on the vocabulary[4] 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[5].

```
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
```

[3] 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.

[4] vocabulary $\sim$ 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.

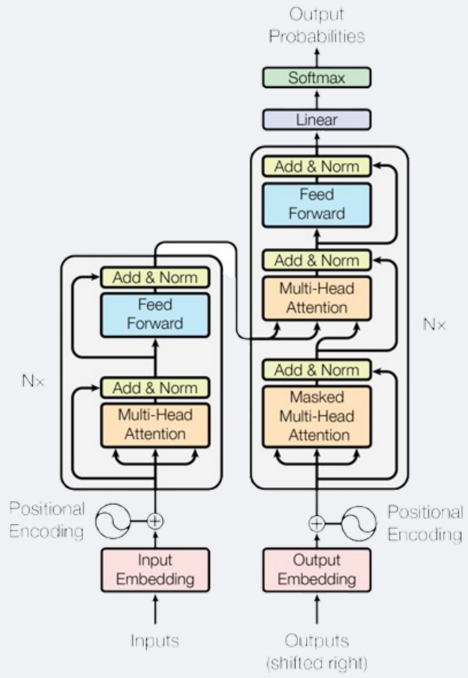
[5] our look-up tables are very much capable of any encoding and decoding (for the tiny tiny vocabulary).

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.
Hoever, when looking into the Transformers model architecture as outlined in the paper[1], also in[6] for convenience, it is seen that the first block is an Embedding block.

[6] Transformers architecture



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[7] 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:

[7] *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 indeces basically same as the position of the single *1s* in the one-hot vector you would have passed to *nn.Linear*

```
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.

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[8] highlights the evolution of positional encoding in transformer models, a worthy read! For this article, let's focus on the rotary positional embedding (RoPE)[9].
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 $\boldsymbol{W}$ 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
$\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]\}$
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"
    )
)
```

[8] https://huggingface.co/blog/designing-positional-encoding
You could have designed state of the art positional encoding
Christopher Fleetwood

[9] arXiv: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

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

Having implemented $\vec{\theta}$, next let's implement $m\vec{\theta}$ by way of an outer product[10]

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

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

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

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

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

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

[10] *context_len* is an integer which refers to the maximum number of tokens the model can consider in a single forward pass

Implementing the rotation mechanism

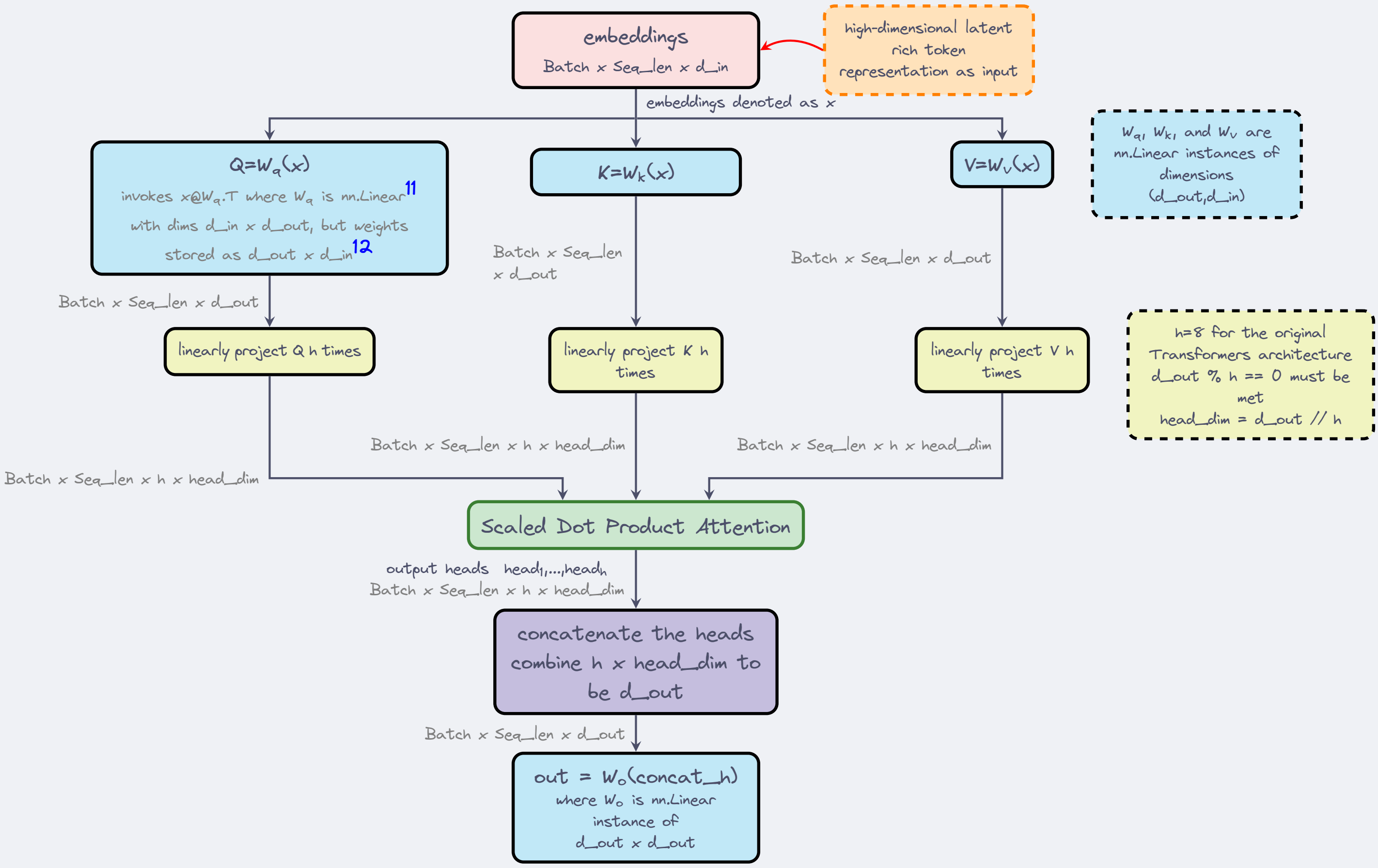
the previously derived mathematical algorithm can then be translated into code as below.

```python
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[13]

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



11 *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*

12 proving the invocation that initializes Q, K and V matrices

```python
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
```

13 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.

...

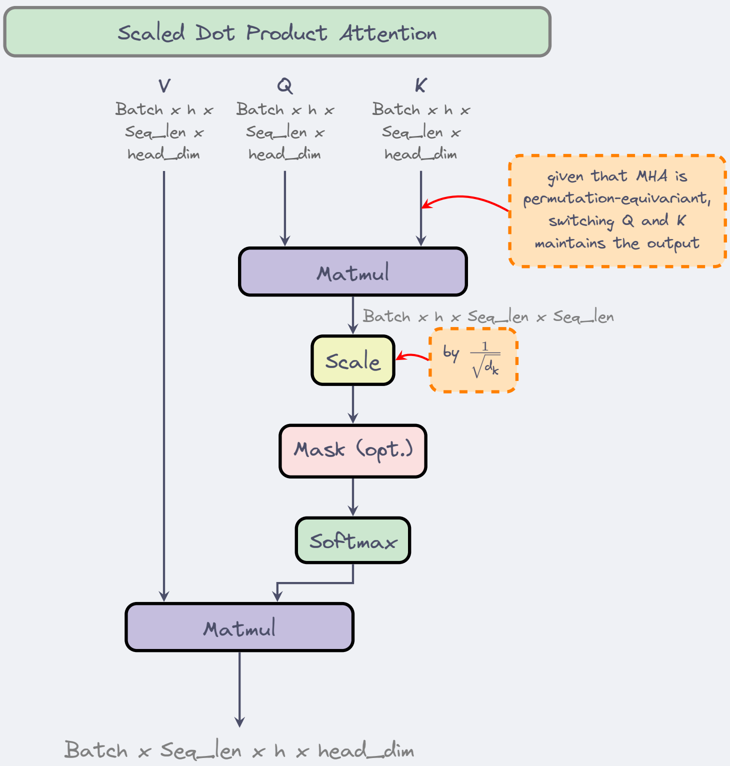
Scaled dot product attention

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[14]. 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 [13]...

- ***values*** ~ for each input element, we also have a value vector. This feature vector is the one we want to average over.
- ***score function*** ~ 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***

[14] 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 x Seq_len x h x head_dim* before this step where d_k is gotten by *k.shape[-1]*

During the decoding phase, at each step of predicting a word[15], 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.

[15] the model actually predicts a token which, by using a lookup table, is decoded to a word which is what humans understand.

Implementing masking in code

Let's use the sequence below
Eiffel Tower is in Paris
and consider the llama 2 tokenizer[16], *sentencepiece*, as the final Transformers model built on these progressive learnings while building on the architecture is Llama 2.

[16] 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

```
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])
```

```
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

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

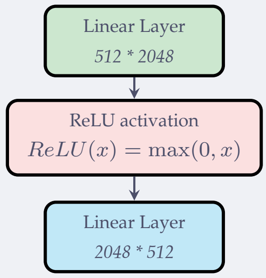
where x is basically the same sequence (as an embedding) input to the $Q, K\&V$. This layer hence is a residual connection necessary for enabling smooth gradient flow through the model and retaining information from the original sequence prior to the multi-head attention. This is simply implemented as

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

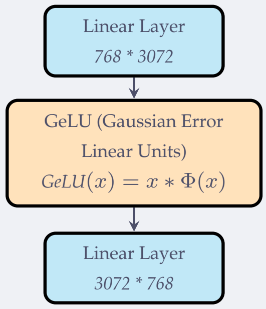
What about the Feed Forward Network layer

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

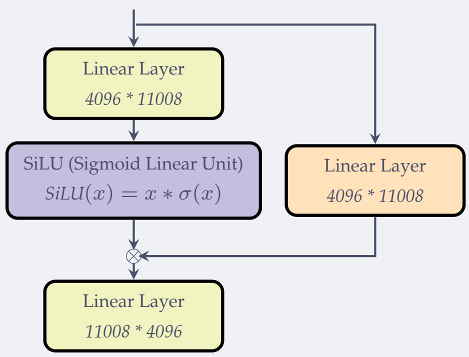
[17] *Feed Forward NN layer for Transformer model*



Feed Forward NN layer for GPT-2



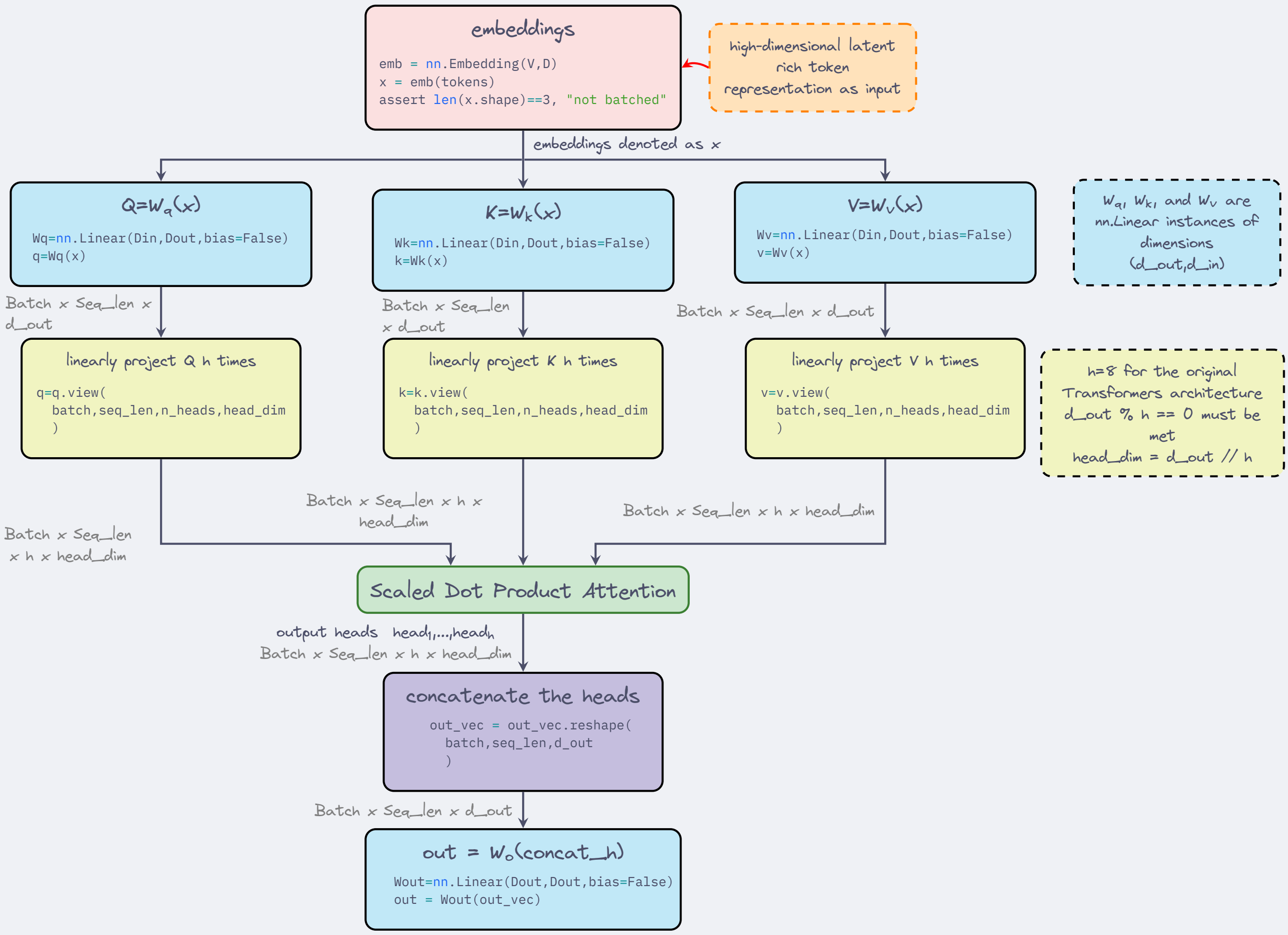
Feed Forward NN layer for Llama-2-7b



[18] disjointed for it does not exist in a nn.Module class with the updatable weights in the init part of the class

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[18] 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
```

As the paper[9] says, "...to any $x_i \in R^d$ where d is even..."

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

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

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

$$\begin{pmatrix} m_1\theta_1, & m_1\theta_2, & \ldots, & m_1\theta_{d/2-1}, & m_1\theta_{d/2} \\ m_2\theta_1, & m_2\theta_2, & \ldots, & m_2\theta_{d/2-1}, & m_2\theta_{d/2} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ m_{cl}\theta_1, & m_{cl-1}\theta_2, & \ldots, & m_{cl-1}\theta_{d/2-1}, & m_{cl-1}\theta_{d/2} \end{pmatrix}$$

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 $(\ldots,d1)$ to $\left(\ldots,\frac{d1}{2},2\right)$ where d1 is even

dims transformed from $(\ldots, d)$ to $\left(\ldots, \frac{d}{2}\right)$

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)[19].

[19] reminder that the rotational transformation is to be applied to the queries and keys only and not the values (*refer to page 4*).

⋆ which is then done below[20]

```
# 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)
```

[20] full neat implementation
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[21] activation function applied between the two linear transformations.

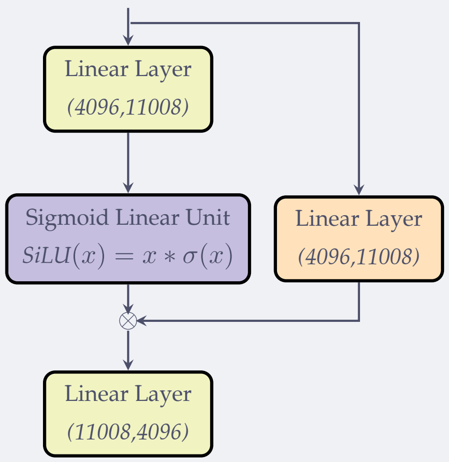
$$FFN(x, W_1, W_2, b_1, b_2) = \max(0, xW_1 + b_1)W_2 + b_2$$

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[22], a variation of the Transformer FFN layer which then uses a variant of the Gated Linear Unit[23]. This leads to the FFN layer having three weight matrices as opposed to the original two which yields the implementation below.

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

[22] https://arxiv.org/abs/2002.05202v1
GLU Variants Improve Transformer
Noam Shazeer 2020

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

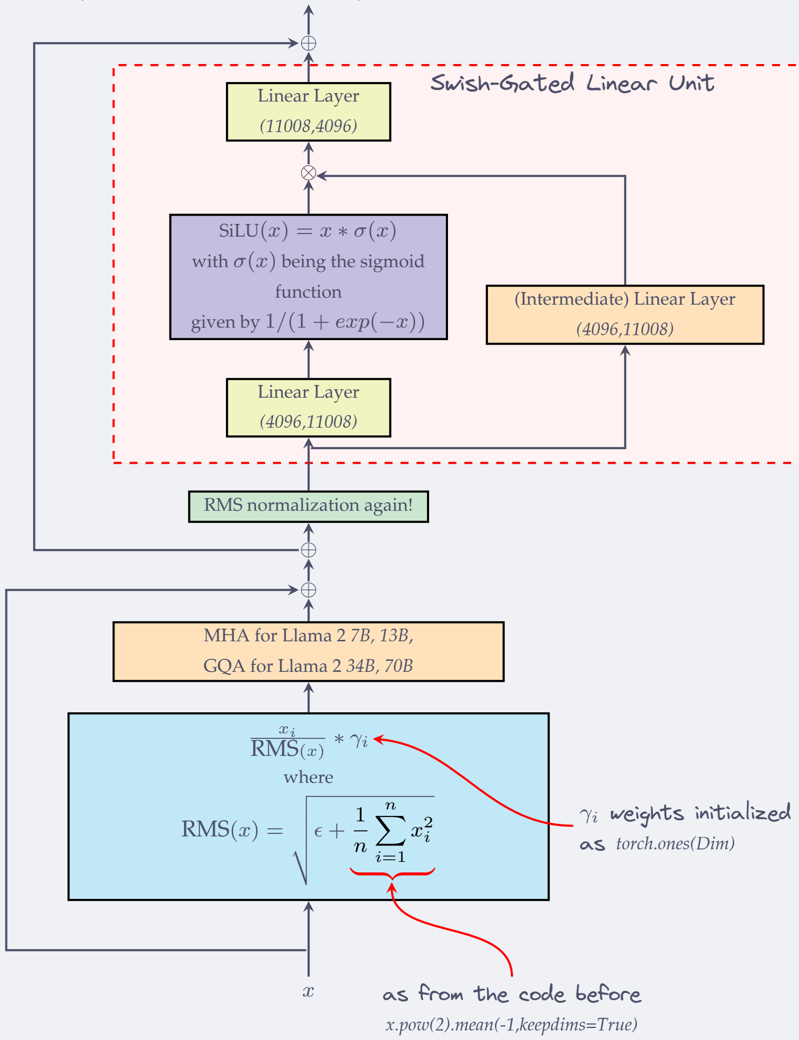



```python
class FeedForward(nn.Module):
  def __init__(self,dim,hidden_dim,bias=False,device="cpu"):
      super(FeedForward,self).__init__()
      self.w1 = nn.Linear(dim,hidden_dim,bias=bias,device=device)
      self.v = nn.Linear(dim,hidden_dim,bias=bias,device=device)
      self.w2 = nn.Linear(hidden_dim,dim,bias=bias,device=device)
      self.SiLU = nn.SiLU()
  def forward(self,x):
      x = (self.SiLU(self.w1(x))*self.v(x))
      x = self.w2(x)
      return x
```

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

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

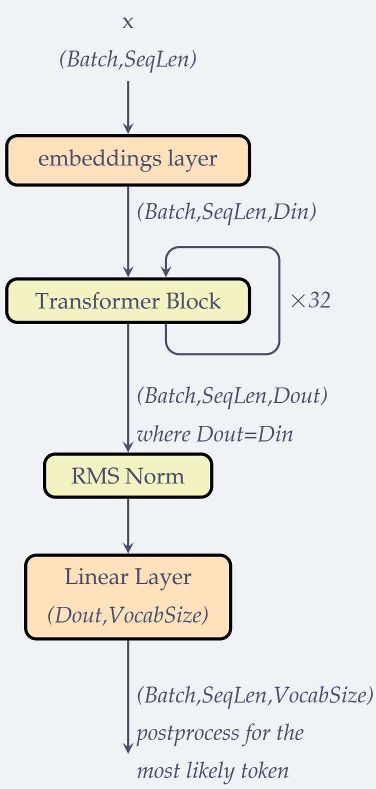


With the above nice input-output mapping translating to code as

```
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.ffn = 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.ffn(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.



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
```

with the model translating to code as

```
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

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[24]

[24] https://modal.com/docs/examples

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

```python
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

```python
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

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

to now being

```python
@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[25], for now going for the T4.

```python
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[26]

```
download_model.remote()
```

[26] 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[27], 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[28] parameters, currently being inferenced at full precision, hence yielding Q as 32, the lower bound for GPU is then

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

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

$$M = 33.6\, GB$$

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[29].

[27] Calculating GPU memory for serving LLMs

[28] 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.

[29]

```
    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[30]. 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[31] being the first of many that stuck in my mind.

This video goes into optimization using Jax[32], 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[33] given different performance metrics quantifying how fast training gets.[34] This precursor study was a huge stepping stone to training Gemma group of models by Google, *Gemma 3n*[35] being my most beloved, given it was focused on optimization and hence inference for relatively lower memory-constrainted 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.

30 https://www.youtube.com/watch?v=vKcA094FSMk
Demo: Gemma 2 architecture: JAX, Flax, and more

31 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

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

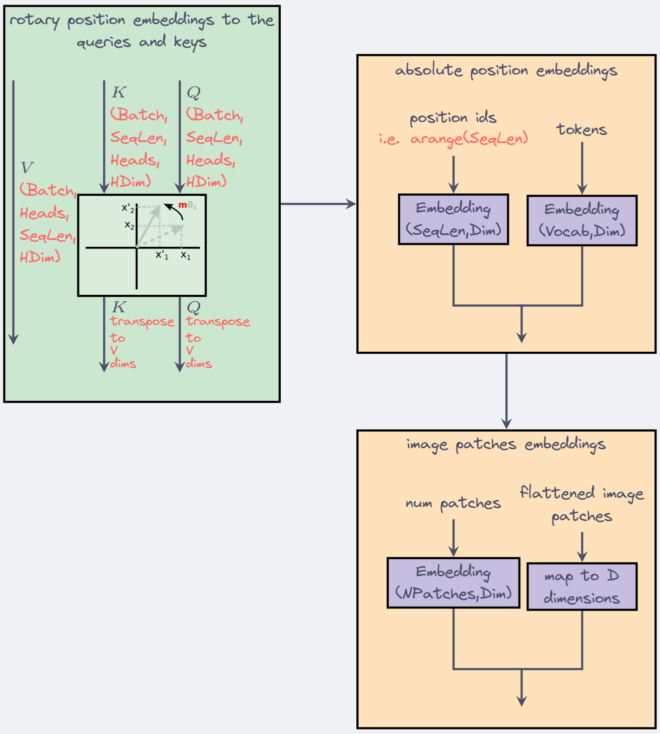
33 arXiv:2010.11929
An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
Dosovitskiy et al. 2020

34 ViT PyTorch vs JAX training benchmarks on Vertex AI Training Platform

35 https://www.youtube.com/watch?v=eJFJRyXEHZ0
Announcing Gemma 3n Preview: Powerful, Efficient, Mobile-First AI

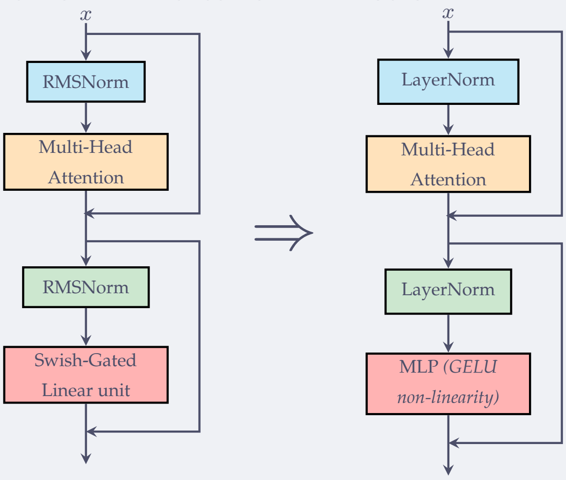
Dumbing down Llama2 to base ViT

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

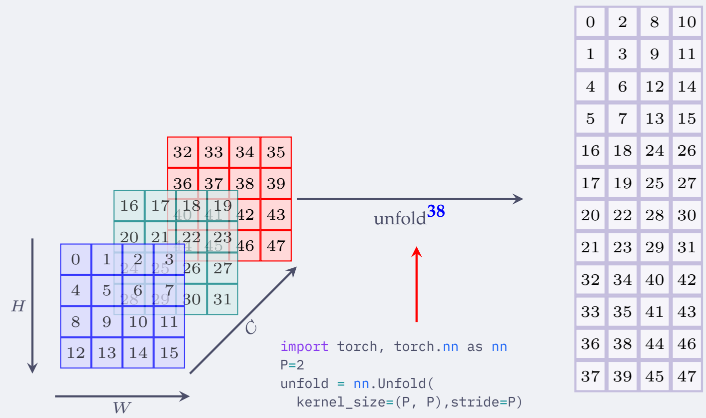
Going onto the interesting part of what makes this pioneering research awesome, is how the image patches is handled. As described in *page 3 section 3.1* of the paper[33], *we reshape the image $x \in R^{H \times W \times C}$ into a sequence of flattened 2D patches $x_p \in R^{N \times (P^2 \cdot C)}$, 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*[36], 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[37].



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*[39] 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.

[36] Unfold
PyTorch API reference

[37] the tensor needs extrusion to the batch dimension since unfold only supports 4-D tensors

[38] 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!

[39] *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])
```

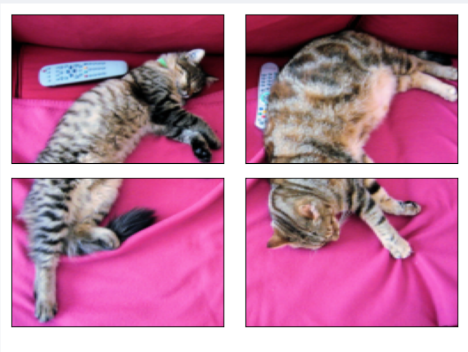
and so the patches can be converted to images as

```
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[40] 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[41]

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

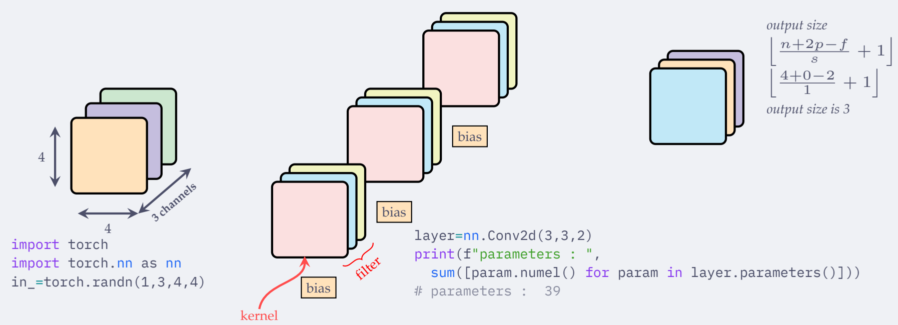
```
from einops.layers.torch import Rearrange
rearr = Rearrange(
    'b c (h p1) (w p2) -> b (c p1 p2) (h w)',
    p1 = P, p2 = P)
```

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.[42]

```
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[43].



[41] https://einops.rocks/pytorch-examples.html *Writing a better code with pytorch and einops*

[42] Note that as we understood from [38], 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 $x_p \in R^{N \times (P^2 \cdot C)}$.

[43] 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[45] to the sequence of embedded patches
$[\mathrm{x}_{\text{class}}; \mathrm{x}_p^1\mathbf{E}; \mathrm{x}_p^2\mathbf{E}; \cdots; \mathrm{x}_p^N\mathbf{E}]$

```
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

[44] 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 $C \times P^2$.

[45] 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

```
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.class_tokens = nn.Parameter(
            torch.randn(1, 1, dim))
        self.position_embeddings = 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)
        x = torch.cat((self.class_tokens,x),dim=1)
        x += self.position_embeddings[:,:(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[46]. 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)
```

[46] 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.

```python
    # ...
    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.

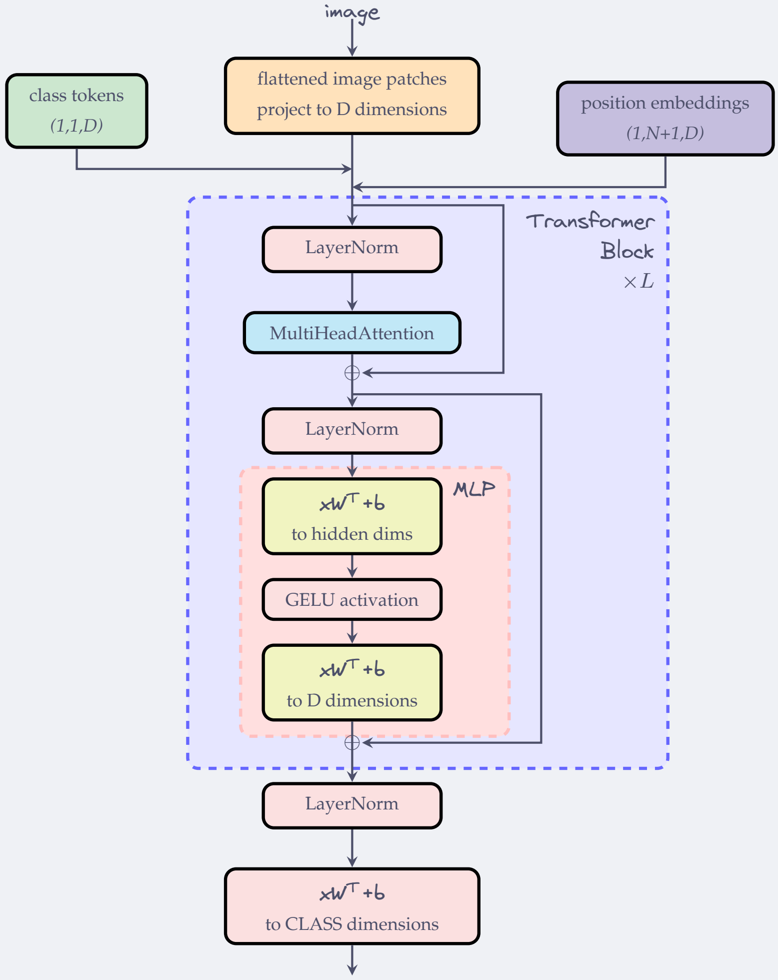
```python
class MLP(nn.Module):
    def __init__(self,dim,hidden_dim,bias=True):
        super(MLP,self).__init__()
        self.fc1 = nn.Linear(dim,hidden_dim,bias=bias)
        self.fc2 = nn.Linear(hidden_dim,dim,bias=bias)
        self.act = nn.GELU(approximate='tanh')
    def forward(self,x):
        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*[47] is beautifully implemented as a PyTorch module as follows

```python
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
```

[47] as shown in the figure at the end of page 21

And with that, the whole model stands out as




```python
class VisionTransformer(nn.Module):
    def __init__(self):
        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 [48]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 [49]

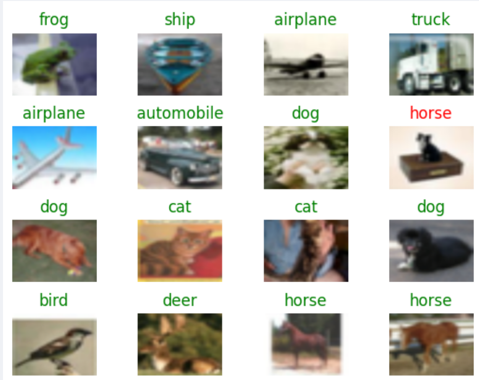
```
vit = VisionTransformer()
weights = torch.load(
    "../models/vit_b_16_pretrained_cifar10.pth",
    weights_only=True, map_location='cpu')
copy_model_weights(vit, weights)
vit = vit.to(device); vit.eval()

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))
])

images, labels = next(iter(test_loader))
for i, (image, label) in enumerate(zip(images,labels)):
    image = image.to(device)
    image = image.unsqueeze(0)
    plt.subplot(4,4,i+1)
    with torch.no_grad():
        pred = vit(image)
    pred = pred.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()
```

48 arXiv:2106.04560
Scaling Vision Transformers
Zhai et al. 2022 (The infamous Lucas Beyer is also a co-author on this)

49 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, *Lorem the capital ipsum*

lorem ip