Scaling ViTs across Training Compute

by Marvin

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.

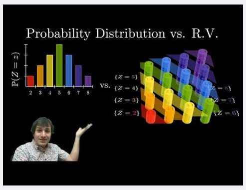
[1] arXiv:1706.03762
Attention is all you need
Vaswani et al. 2017

[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.

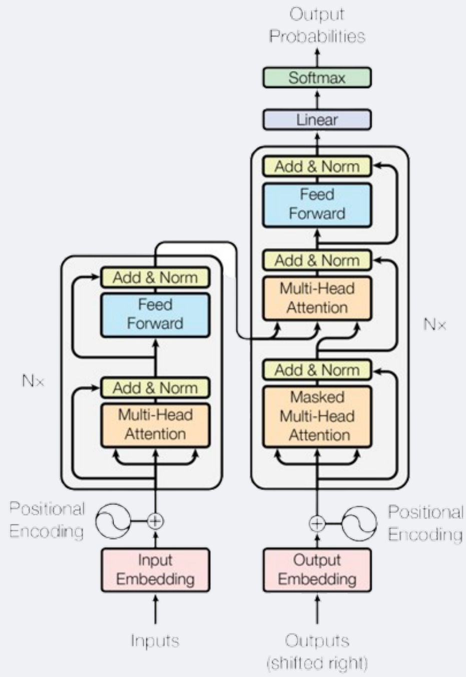
What about the Embeddings block?

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:

```
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.

[6] Transformers architecture



[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*

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()
```

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

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

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

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

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

$$= (x_1 + ix_2)(f_{11} + i\hat{f}_{11}) \;=\; x_1 f_{11} - x_2\hat{f}_{11} + i(x_1\hat{f}_{11} + x_2 f_{11})$$

$$\text{meaning} \begin{pmatrix} x_1 + ix_2 \\ x_3 + ix_4 \end{pmatrix} \otimes \begin{pmatrix} f_{11} + i\hat{f}_{11} \\ f_{12} + i\hat{f}_{12} \end{pmatrix}$$

$$= \begin{pmatrix} x_1 f_{11} - x_2\hat{f}_{11} + i(x_1\hat{f}_{11} + x_2 f_{11}) \\ x_3 f_{12} - x_4\hat{f}_{12} + i(x_3\hat{f}_{12} + x_4 f_{12}) \end{pmatrix} = \begin{pmatrix} (x_1 f_{11} - x_2\hat{f}_{11} & x_1\hat{f}_{11} + x_2 f_{11}) \\ (x_3 f_{12} - x_4\hat{f}_{12} & x_3\hat{f}_{12} + x_4 f_{12}) \end{pmatrix}$$

$$\text{rearranging gives}$$

$$= \begin{pmatrix} x_1 f_{11} - x_2\hat{f}_{11} \\ x_1\hat{f}_{11} + x_2 f_{11} \\ x_3 f_{12} - x_4\hat{f}_{12} \\ x_3\hat{f}_{12} + x_4 f_{12} \end{pmatrix} \Rightarrow \begin{pmatrix} x_1\cos m_1\theta_1 - x_2\sin m_1\theta_1 \\ x_1\sin m_1\theta_1 + x_2\cos m_1\theta_1 \\ x_3\cos m_1\theta_2 - x_4\sin m_1\theta_2 \\ x_3\sin m_1\theta_2 + x_4\cos m_1\theta_2 \end{pmatrix}$$

[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

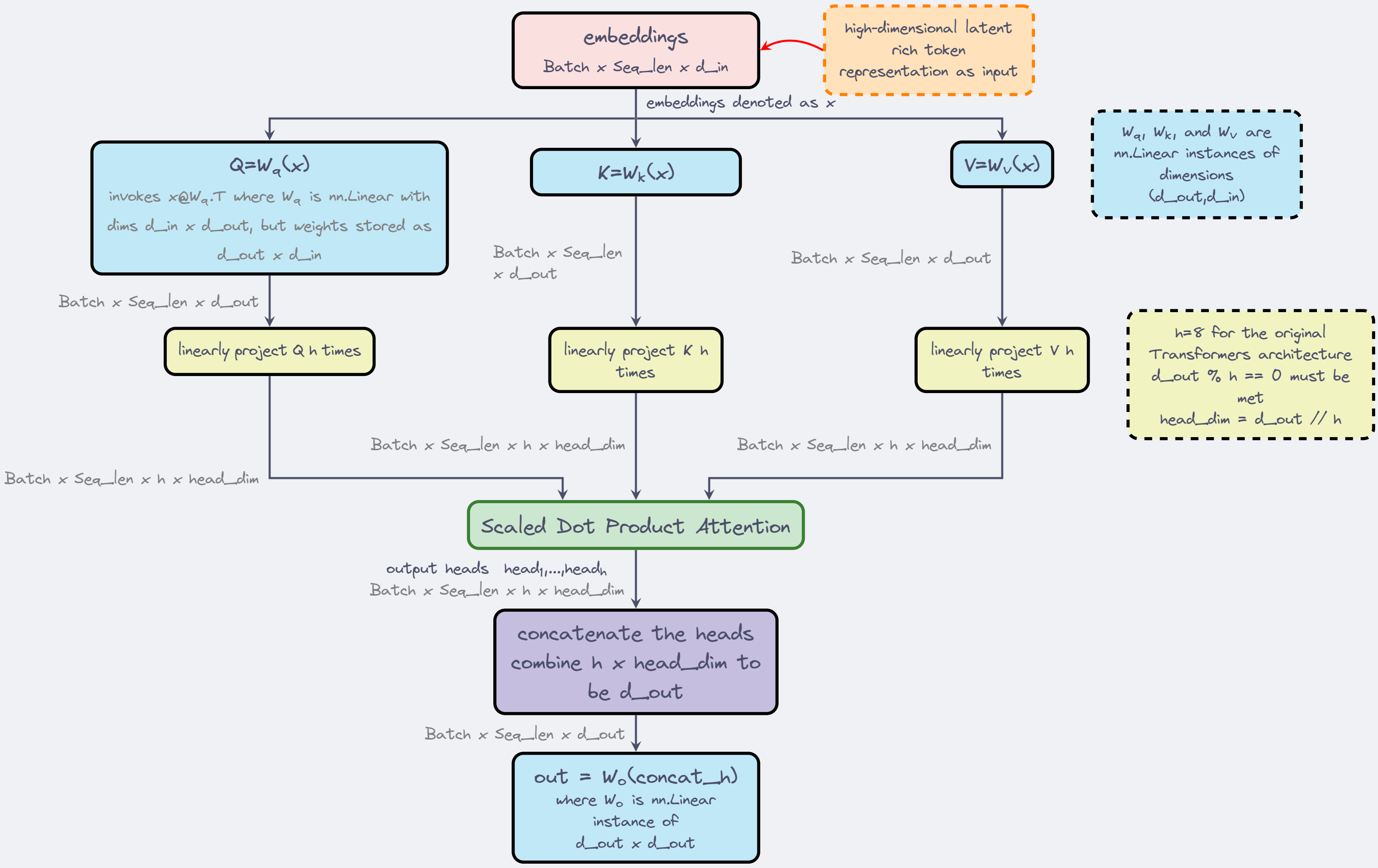
the previously derived mathematical algorithm can then be translated into code as below.

```
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

```
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.

. . .

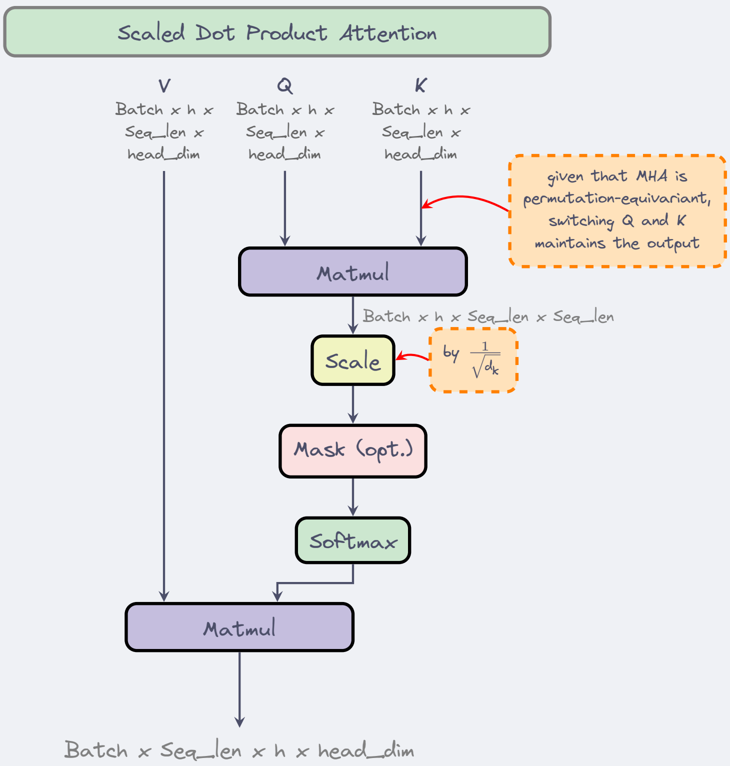
Scaled dot product attention

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

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

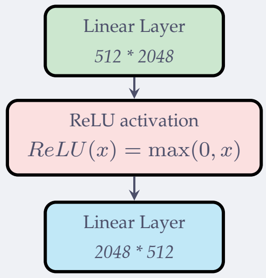
where x is basically the same sequence (as an embedding) input to the $Q, K\&V$. This layer hence is a residual connection necessary for enabling smooth gradient flow through the model and retaining information from the original sequence prior to the multi-head attention. This is simply implemented as

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

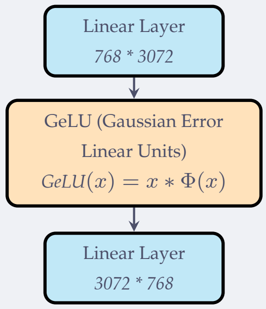
What about the Feed Forward Network layer

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.

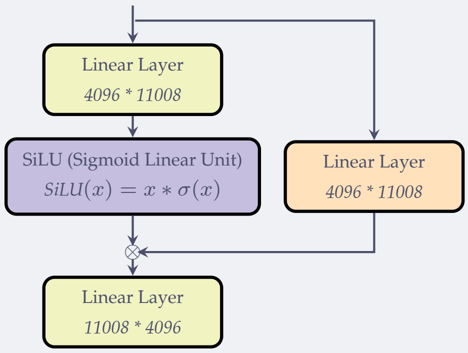
[17] *Feed Forward NN layer for Transformer model*



Feed Forward NN layer for GPT-2



Feed Forward NN layer for Llama-2-7b

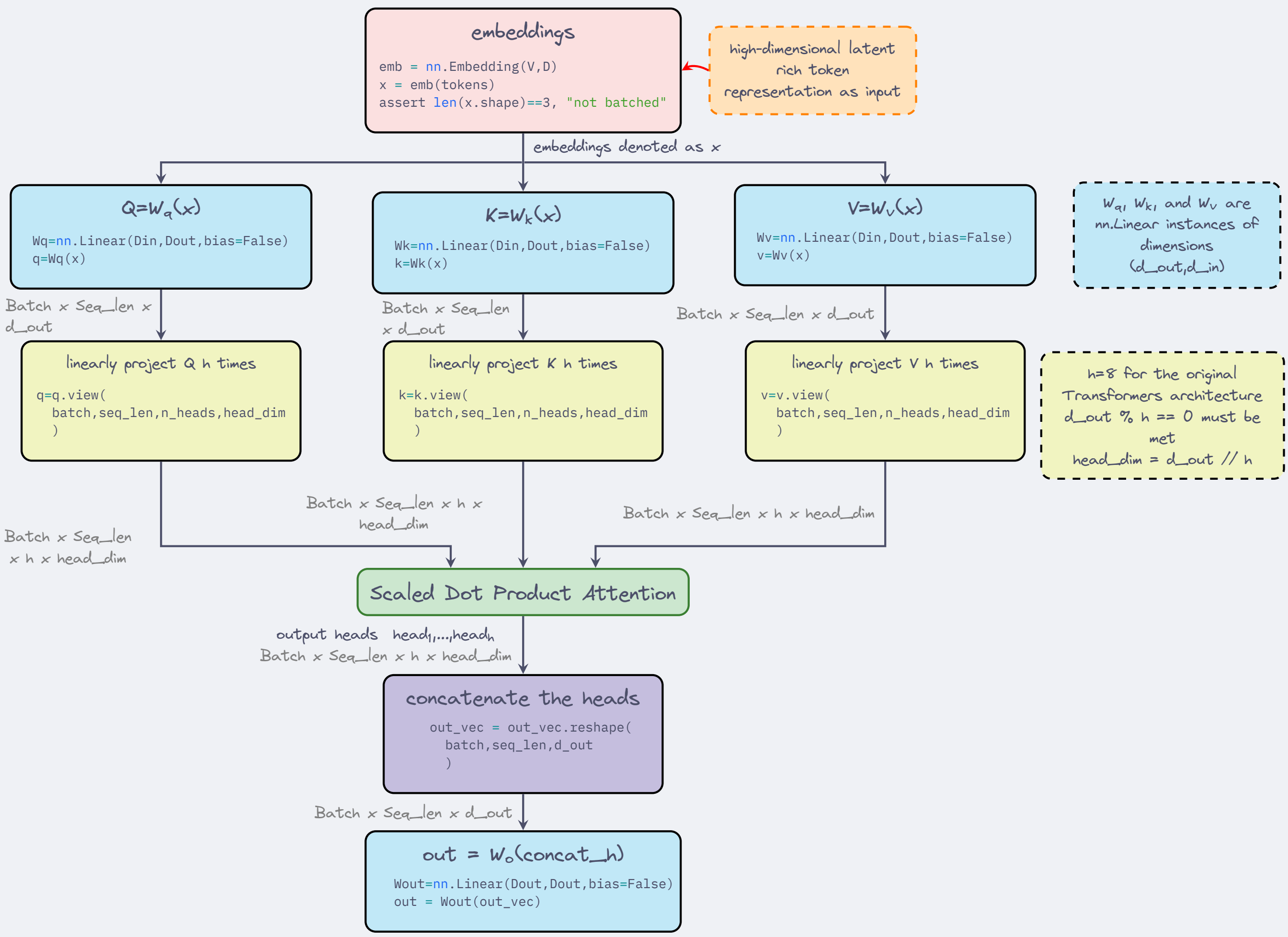


Building Llama-2 from the SDPA outwards...

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?

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

Recap on Rotational Positional Encoding

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

```
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, in the graph for the MultiHead Attention, let's focus on the part after the transformation to the dimensions

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

but before the high-dimensional transpose to get the batch of heads each with dimensions (seq_len,head_dim)[19].

```
# 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)
```

The neat implementation can be found here[20]

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

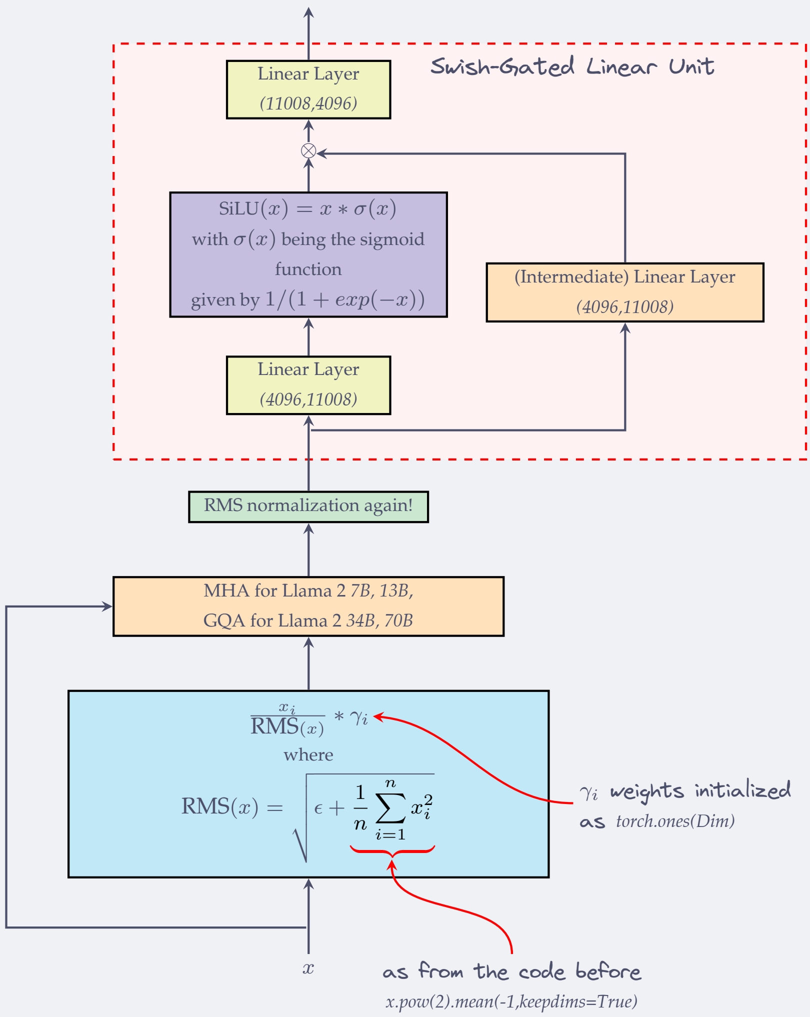
```
class RMSNorm(torch.nn.Module):
    def __init__(self, dim: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))
    def forward(self, x):
        means = x.pow(2).mean(-1, keepdim=True)
        norm_x = x * torch.rsqrt(means + 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 by few as SwiGLU, a Swish-Gated Linear Unit[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. Hence, from these clarifications, then the Transformer block is visualized as 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

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
```

NEW PAGE PENDING.......

lorem ipsum