RQ: What impact do different layer normalisation methods (Pre-LN vs. Pre-PN vs. Pre-RMSN) have on the training effectiveness and performance of decoder-only transformer architectures (e.g. GPT-2)?

(100 words)

Section 1 — Introduction - Deep Learning and Decoder-only Transformers

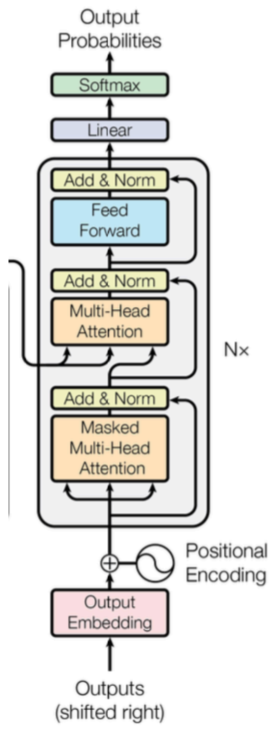


figure 1

Deep-learning (DL), a specialised subset of machine-learning (ML), relies on sophisticated optimizer algorithms to discern complex patterns upon training on vast datasets. Unlike traditional ML which utilises simpler architectures, DL employs more layers of neural networks, enabling it to capture and process even more complex data structures with minimal human intervention.

A recent DL architecture—decoder-only transformers, had been profoundly important for the domain of NLP (natural-language-processing)—notably led to the creation of sophisticated chatbots like ChatGPT. Captivated by its inherent intelligence, my research aims to investigate potential architectural improvements in transformers, potentially improving its performance and reducing training cost.

(400 words)

Section 2 — Background Information - How decoder-only transformers work

In this section, I will illustrate how the transformer type which I'm working with (a decoder-only transformer) works: with the example of trying to complete a sentence. Let's take the example:

$$\text{“\textit{The brown fox jumps over the lazy dog}”} \tag{1}$$

Currently, we have only prompted our transformer with:

$$\text{“\textit{The brown fox jumps}”} \tag{2}$$

And given prompt (1), I will show you how does the transformer generate text and completes the sentence to achieve (2)!

Tokenizer Embedding

Since neural networks only work with numbers (high-dimensional vectors to be precise), and we need transformers to operate on natural language (English). We need tokenization! Which converts words into high-dimensional vectors.

The tokenizer first converts the prompt (2) into a list of IDs, which the tokenizer encoder as seen in Figure-1 converts the IDs to embedding vectors, which contains the semantic meaning (learnt during training) of the word in high-dimensional space. This process can be seen below:

$$\begin{bmatrix} The \\ brown \\ fox \\ jumps \end{bmatrix} \xrightarrow[tokenizer]{} \begin{bmatrix} 101 \\ 2078 \\ 513 \\ 778 \end{bmatrix} \xrightarrow[layer]{embedding} \begin{bmatrix} [\;0.21 & -1.03 & 0.87 & 0.58 & -0.76\;] \\ [-0.45 & 0.92 & -0.23 & 1.12 & 0.09\;] \\ [\;0.98 & -0.14 & 0.45 & -0.88 & 0.63\;] \\ [-0.67 & 0.73 & 0.36 & 0.21 & -0.47\;] \end{bmatrix} \tag{3}$$

Note: This is an oversimplification, as we are only showing a $d_{embd} = 5$.

Positional Embedding

Since in English, the order of words matter in a sentence. We need a way for the transformer to learn about the positional relationship of the different tokens. To do this, (Google Inc #) proposes the positional embedding, following:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{4}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{5}$$

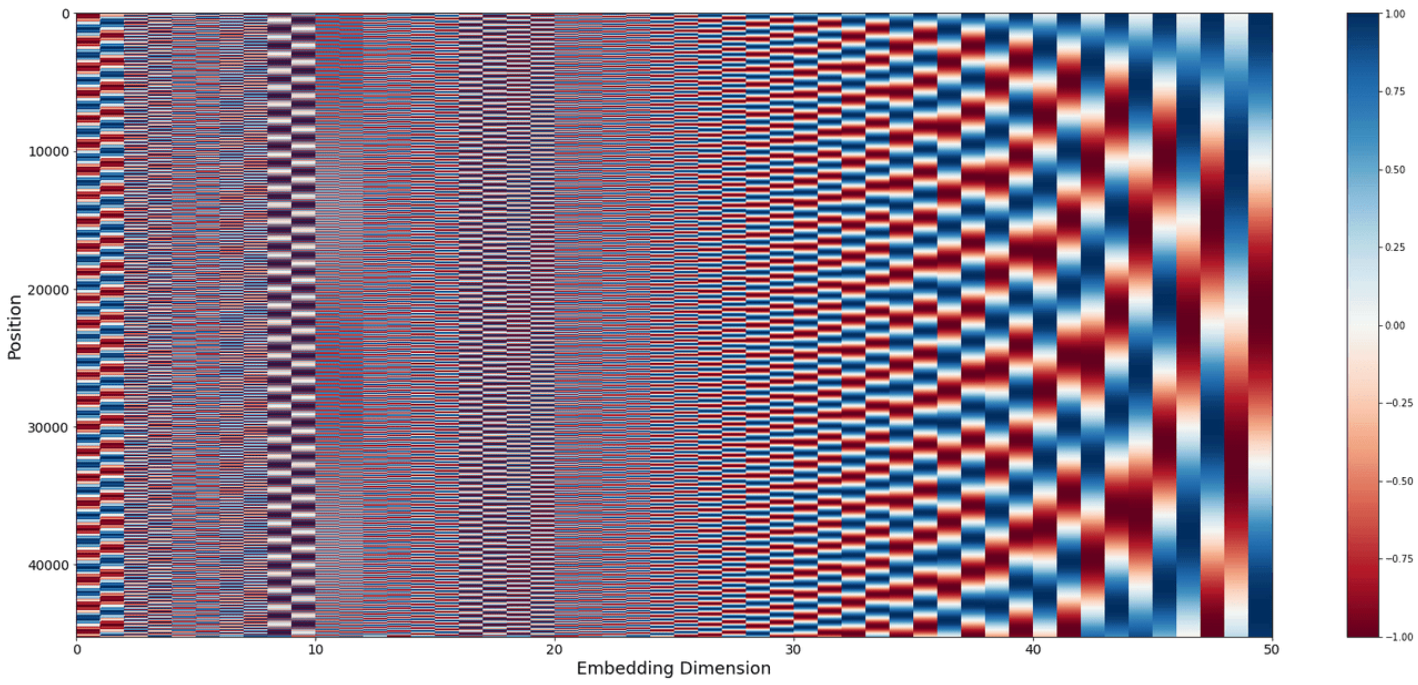


Figure-2: Graphing out Positional Embedding (4), (5)

Which, we can visualise more clearly how the positional embedding (4), (5) applies for our prompt (2):

$$\begin{bmatrix}[pos=0]\\ [pos=1]\\ [pos=2]\\ [pos=3]\end{bmatrix} \xrightarrow[encoder]{positional} \begin{bmatrix}[PE(0,0) & PE(0,1) & PE(0,2) & PE(0,3) & PE(0,4)]\\ [PE(1,0) & PE(1,1) & PE(1,2) & PE(1,3) & PE(1,4)]\\ [PE(2,0) & PE(2,1) & PE(2,2) & PE(2,3) & PE(2,4)]\\ [PE(3,0) & PE(3,1) & PE(3,2) & PE(3,3) & PE(3,4)]\end{bmatrix}$$

$$= \begin{bmatrix}\left[\sin\left(\frac{0}{10000^{0/5}}\right) & \cos\left(\frac{0}{10000^{0/5}}\right) & \sin\left(\frac{0}{10000^{2/5}}\right) & \cos\left(\frac{0}{10000^{2/5}}\right) & \sin\left(\frac{0}{10000^{4/5}}\right)\right]\\ \left[\sin\left(\frac{1}{10000^{0/5}}\right) & \cos\left(\frac{1}{10000^{0/5}}\right) & \sin\left(\frac{1}{10000^{2/5}}\right) & \cos\left(\frac{1}{10000^{2/5}}\right) & \sin\left(\frac{1}{10000^{4/5}}\right)\right]\\ \left[\sin\left(\frac{2}{10000^{0/5}}\right) & \cos\left(\frac{2}{10000^{0/5}}\right) & \sin\left(\frac{2}{10000^{2/5}}\right) & \cos\left(\frac{2}{10000^{2/5}}\right) & \sin\left(\frac{2}{10000^{4/5}}\right)\right]\\ \left[\sin\left(\frac{3}{10000^{0/5}}\right) & \cos\left(\frac{3}{10000^{0/5}}\right) & \sin\left(\frac{3}{10000^{2/5}}\right) & \cos\left(\frac{3}{10000^{2/5}}\right) & \sin\left(\frac{3}{10000^{4/5}}\right)\right]\end{bmatrix}$$

$$= \begin{bmatrix}[0.000 & 1.000 & 0.000 & 1.000 & 0.000]\\ [0.841 & 1.000 & 0.001 & 1.000 & 0.000]\\ [0.909 & 0.999 & 0.001 & 1.000 & 0.000]\\ [0.141 & 0.997 & 0.002 & 1.000 & 0.000]\end{bmatrix} (3\ d.p.) \tag{6}$$

Hence, from (6), we can see an example of how positional embedding (4), (5) generates unique positional embeddings for each text in a given position! As seen in Figure-2, (4), (5) actually ensures that no matter how long, each token/word in a sequence actually has its own unique embedding!

Before passing into the decoder block, Positional Embedding and Token Embedding of (2) is then added, so that our token contains both the semantic meaning and positional information of words in (2):

$$Embedding\ Summation\ =\ Token\ Embedding\ +\ Positional\ Embedding \tag{7}$$

Our prompt:

$$\begin{bmatrix}[0.000 & 1.000 & 0.000 & 1.000 & 0.000]\\ [0.841 & 1.000 & 0.001 & 1.000 & 0.000]\\ [0.909 & 0.999 & 0.001 & 1.000 & 0.000]\\ [0.141 & 0.997 & 0.002 & 1.000 & 0.000]\end{bmatrix} + \begin{bmatrix}[0.21 & -1.03 & 0.87 & 0.58 & -0.76]\\ [-0.45 & 0.92 & -0.23 & 1.12 & 0.09]\\ [0.98 & -0.14 & 0.45 & -0.88 & 0.63]\\ [-0.67 & 0.73 & 0.36 & 0.21 & -0.47]\end{bmatrix}$$
$$= \begin{bmatrix}[0.21 & -0.03 & 0.87 & 1.58 & -0.76]\\ [0.39 & 1.92 & -0.23 & 2.12 & 0.09]\\ [1.89 & 0.86 & 0.45 & 0.12 & 0.63]\\ [-0.53 & 1.73 & 0.36 & 1.21 & -0.47]\end{bmatrix} \quad (8)$$

Great! Now that our input contains both semantic meaning and positional information, we can start processing it to predict the next word!

Layer Normalisation (56 words)

Layer normalisation, as seen in Figure-1, is a normalisation operation applied to our inputs before passing onto operations such as Attention-Mechanism and Feed-Forward Network. It ensures that the activations remain stable across layers, preventing issues such as internal covariate shift.

Since this is our main exploration, we will go into this in more depth in a later section.

Q, K, V, and Attention (78 words)

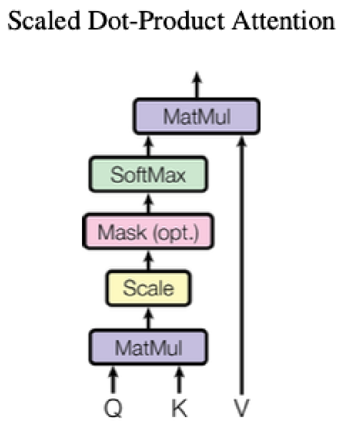


Figure-3: Scaled Dot-Product Attention

$$Attention(Query,\ Key,\ Value) \ = \ softmax\left(\frac{Query \bullet Key^{Temperature}}{\sqrt{dimensionality\ of\ key\ vectors}}\right) \bullet Value \quad (9)$$

First, we need to pass our embeddings into the “Multi-Head attention” which contains multiple “Self-Attention mechanism”:

How does self-attention work?

1. $Query,\ Key,\ Value$ matrices are derived from multiplying $Embedding\ Summation$ to learn matrices.
   a. $Query$: It's like a searchlight pointing at something we're interested in.
   b. Key: It's like labels on items we're searching through.
   c. $Value$: Holds the content of each item.

2. $Query \bullet Key^{Temperature}$, QK^{T}: This operation determines how relevant each item's label (Key) is to our searchlight ($Query$).
3. QK^{T} is scaled by $\frac{1}{\sqrt{dimensionality\ of\ key\ vectors}}$, $\frac{1}{\sqrt{d_k}}$ ($\frac{1}{\sqrt{5}}$ in our case) to ensure numerical stability during learning.[1]
4. $softmax()$ normalises these value to $[0,\ 1]$, effectively selecting the input parts to focus on.
5. Multiplying $softmax(\frac{QK^{T}}{\sqrt{d_k}})$ by $Value$: Forms a weighted sum, here based on the relevance of each items—QK^{T}, their respective content ($Value$) are emphasised.

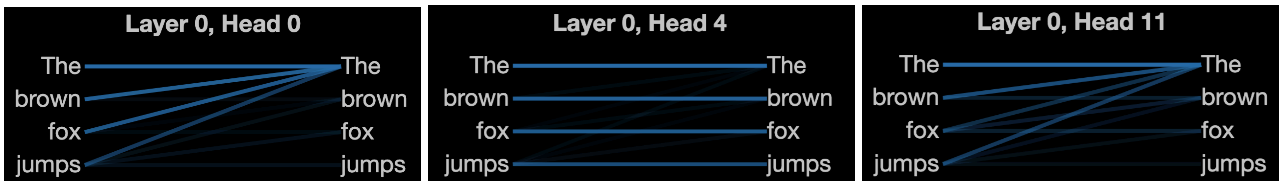


Figure-4: Visualisation of GPT-2's self-attention given the input "The brown fox jumps" using BertViz[2]

Basically, this operation tells the model to pay attention to more contextually significant words. As seen in Figure-4, given our prompt (2), for all words, more "*attention*" is placed on the word "The".

The results from the self-attention heads' (*as seen in Figure-3, each "Multi-head Attention" layer having multiple "Self-Attention heads", GPT-2 has 12 heads for example*) are then aggregated, hence "Multi-head attention":

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

Figure-4: Multi-head Attention formula from "Attention is All you need" paper[3]

Which, let's visualise what happens with our $embedding\ summation$:

$$\begin{bmatrix}\begin{bmatrix} 0.21 & -0.03 & 0.87 & 1.58 & -0.76 \\ 0.39 & 1.92 & -0.23 & 2.12 & 0.09 \\ 1.89 & 0.86 & 0.45 & 0.12 & 0.63 \\ -0.53 & 1.73 & 0.36 & 1.21 & -0.47 \end{bmatrix}\end{bmatrix} \xrightarrow[\text{multi-head}]{\text{attention}} \begin{bmatrix}\begin{bmatrix} 0.18 & -0.02 & 0.68 & 1.37 & -0.66 \\ 0.07 & 0.18 & -0.02 & 0.32 & 0.01 \\ 0.19 & 0.05 & 0.03 & 0.12 & 0.04 \\ -0.09 & 0.31 & 0.02 & 0.23 & -0.06 \end{bmatrix}\end{bmatrix} \quad (10)$$

From (10), you can hence see how the first row, which represents the word "The" retains a heavier weighting than the other 3 rows, which represents "$brown\ fox\ jumps$". Now, our model will pay more attention to "The" during processing, since it has a heavier weight!

[1] (Google Inc #) As mentioned by XX. This is crucial as it to prevents very large values of the dot products from zeroing out gradients during backpropagation
[2] (jessevig)
[3] (Google Inc #)

Feedforward Networks (54 words)

Once normalised by LN, our matrices is passed through the feedforward layer

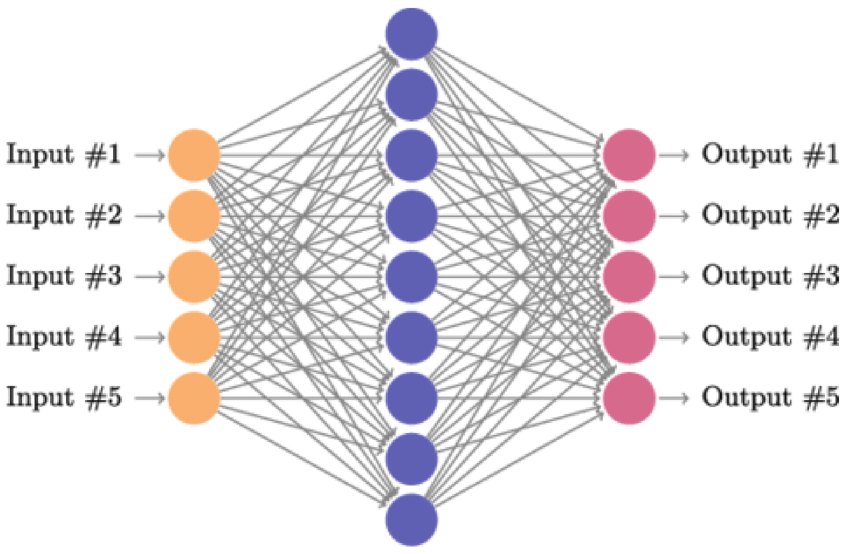


Figure-4: Diagram of Feedforward Layer

BRIEF EXPLANATION HERE OF WHAT DOES feedforward layer DOES, AS IN the idea is to enable the transformer, during training to capture more complex relationships in theory.

Section 3 — Research Question

Now that we are familiar with how the transformer(GPT) works, I will now put forth with my research question:

What impact do different layer normalisation methods (Pre-LN vs. Pre-PN vs. Pre-RMSN) have on the training effectiveness and performance of decoder-only transformer architectures (e.g. GPT-2)?

More precisely:

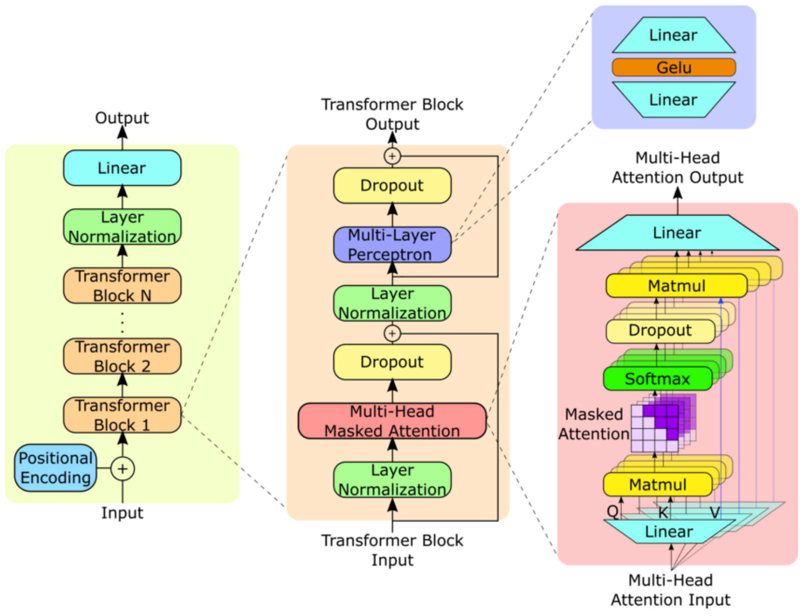


Figure-5: Diagram of GPT2 Architecture

We will investigate how different types of LN compare through re-implementing OpenAI's GPT2 in PyTorch, then train it from scratch. As seen in Figure-5, we can see which part we are investigating—"Layer Normalization".

By default, the 2017 Attention Paper and Open AI's ChatGPT2 had used Pre-LN. The "pre" is referring to the fact that our inputs first passes through "Layer-Normalization" before being passed on to main operations of the transformer (Attention, MLP). Whilst, "LN" refers to Layer Normalisation.

Section 4 — Theoretical Analysis of pre-LayerNormalisation (pre-LN), pre-RootMeanSquareNormalisation (pre-RMSN) and pre-PowerNormalisation (pre-PN)

In this section, we will go through the mathematics behind these 3 types of normalisation.

Section 4.1 — Layer Normalisation

How does Layer Normalisation work?
Here's how it works:

$$y_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \qquad (11)$$

Note: Layer normalisation is applied for every ith row, hence x_i represents each row. For our exemplar below, we will only use $x_0 = [\begin{matrix} 0.18 & -0.02 & 0.68 & 1.37 & -0.66 \end{matrix}]$ *for simplicity. We are also going into more detail here, since Layer Normalisation is the focus of our research.*

pre-LN in Action

Operations of (11)	Sample Calculation
1. μ_i = mean of x_i is calculated with: $\mu_i = \frac{1}{d_k} \sum_{0}^{d_k - 1} x_{(i,n)}$	*Since we are using* $x_0 = [\ 0.18\ \ -0.02\ \ 0.68\ \ 1.37\ \ -0.66]$ *only:* $u_0 = \frac{1}{5}(0.18 - 0.02 + 0.68 + 1.37 - 0.66)$ $= 0.31$
2. σ^2 = the variance x_i is calculated with: $\sigma^2 = \frac{1}{d_k - 1} \times \sum_{0}^{d_k} (x_{(i,n)} - \mu_i)^2$	$\sigma^2 = \frac{1}{5-1}((0.18 - 0.31)^2 + \ldots\ldots + (-0.66 - 0.31)^2)$ $= \frac{1}{4}((-0.13)^2 + (-0.33)^2 + \ldots + (-0.97)^2)$ $= 0.5818$
3. $x_i - \mu_i$: This operation shifts x_i to have a mean of 0.	$x_0 - \mu_0 = [x_{0,0} - \mu_0 \quad x_{0,1} - \mu_0 \quad x_{0,2} - \mu_0 \quad x_{0,3} - \mu_0 \quad x_{0,4} - \mu_0]$ $= [0.18 - 0.31 \quad -0.02 - 0.31 \quad 0.68 - 0.31 \quad 1.37 - 0.31 \quad -0.66 - 0.31]$ $= [-0.13 \quad -0.33 \quad 0.37 \quad 1.06 \quad -0.97]$ *Note: Calculating x_0's mean now results in 0.*
4. $\frac{x_i - \mu_i}{\sqrt{\sigma_i^2}}$: Our input (with mean now equal 0), is divided by its sample standard deviation σ_i. This scales down (or up) the values in each row in our input to have $\sigma^2 (variene)$ of 1.	$\frac{x_i - \mu_i}{\sqrt{\sigma_i^2}} = \left[\frac{x_{0,0} - \mu_0}{\sqrt{\sigma_0^2}}, \quad \frac{x_{0,1} - \mu_0}{\sqrt{\sigma_0^2}}, \quad \frac{x_{0,2} - \mu_0}{\sqrt{\sigma_0^2}}, \quad \frac{x_{0,3} - \mu_0}{\sqrt{\sigma_0^2}}, \quad \frac{x_{0,4} - \mu_0}{\sqrt{\sigma_0^2}}\right]$ $= \left[\frac{-0.13}{0.7631} \quad \frac{-0.33}{0.7631} \quad \frac{0.37}{0.7631} \quad \frac{1.06}{0.7631} \quad \frac{-0.97}{0.7631}\right]$ $= \left[-0.17 \quad -0.43 \quad 0.48 \quad 1.39 \quad -1.27\right] (3\ d.p.)$ *Note: Calculating x_0's σ^2 now results in 1.*
5. But, normally, ϵ (a small value) is added to the scaling function ($\sqrt{\sigma_i^2 + \epsilon}$). So that we don't risk dividing dividing by 0 in the event when $\sqrt{\sigma^2}$ approaches 0. This avoids instability during training.	*Why is* ϵ useful? *Assuming in the event which our* σ^2 *approaches* 0*:* - *Without* ϵ: $\lim_{\sigma^2 \to 0} \left(\frac{x_i - \mu_i}{\sqrt{\sigma^2}}\right) = \frac{x_i - \mu_i}{0}$ $= \infty$ - ∞ is terrible, because this will lead to exploding gradients during backpropagation. This destabilises the training run, causing the model to fail to converge, or even crash! - *With* ϵ *(for GPT-2,* $\epsilon = 10^{-8}$*):* $\lim_{\sigma^2 \to 0} \left(\frac{x_i - \mu_i}{\sqrt{\sigma^2 + 10^{-8}}}\right) = finite\ but\ potentially\ large$ - As seen here, the exploding gradients error has been subverted. Preventing risk of training run failure.

Internal Covariate Shift

Why is Layer Normalisation needed?

The reason we shift μ to 0 and σ^2 to 1, is to address the issue of internal covariate shift. This issue occurs as the distribution of the $Feature\ Vectors$[4] changes during training after being transformed by operations in each layer such as "Multi-head attention" or "Feedforward".

To better visualise this issue, we will assume that each time the $Feature\ Vectors$ is processed by a $Block$[5] without $Layer\ Normalisation$, its μ is shifted by $+\ 1.5$, and its σ^2 is scaled by $\times\ 1.2$. And that $Feature\ Vectors$ will be passed through a transformer with 12 $Blocks$[6].

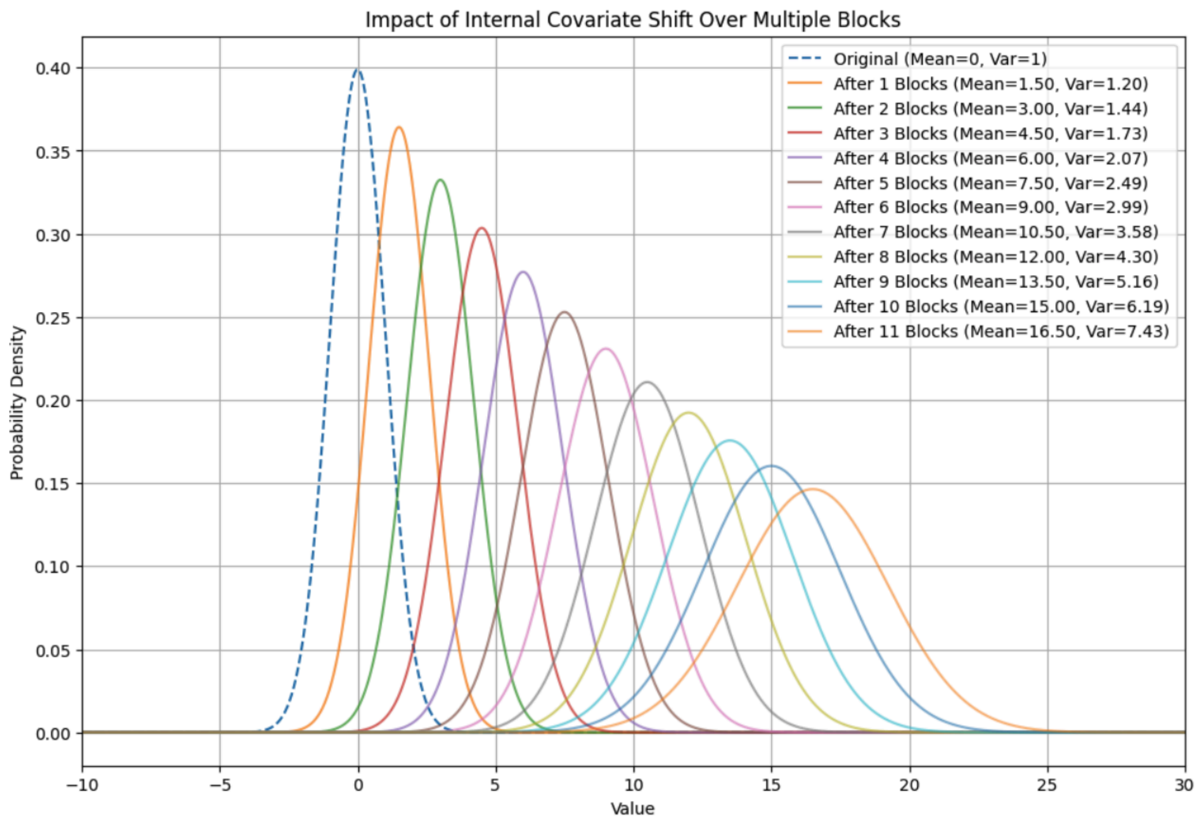


Figure-6: Distribution of $Feature\ Vectors$ after passing through Nx blocks.

As seen in Figure-4, assuming our $Feature\ Vectors$ originally has $\mu\ =\ 0$ and $\sigma^2\ =\ 1$. We can see how rapidly the distribution gets out of hand! After passing through *11*x $Blocks$, our $Feature\ Vectors'$—the input of the *12*th $Block$—distribution had gone completely out of hand! Now having $\mu\ =\ 11$ and $\sigma^2\ =\ 7.43$!

[4] This refers to the data that passes through each layer of the transformer, like the tensor *(10).*

[5] Refer to Figure-1, a block is the section within Nx, it contains processed such as Attention, Layer Normalisation and Feedforward. For example, GPT2-124m contains 12 blocks.

[6] Decided to use this as a baseline because it's the configuration of GPT2-124m, one of the earliest models. Every successive models are a lot bigger than this, with GPT4-06/13 having 120 blocks! Not to mention it has far more heads in its Attention mechanism, which means the mean and variance of *Feature Vectors* will be shifted far more. Hence, demonstrating the importance of reducing *Internal Covariate Shift.*

Why is Internal Covariate Shift bad?

As seen in Figure-4, the accumulation of shifts in data distributions at each layer results in a compounded impact on the network, as the variance is scaled exponentially. Theoretically, this results in[7]:

- **Increased Training Cost**: Early layer's shifted outputs cause subsequent layers to struggle with inconsistent inputs. This forces layers to continually recalibrate their weights and biases, which for each layer needs to be optimised for varying data distributions. This constant recalibration increases computational costs.
- **Reduced Performance:** These inconsistencies likely also impair the network's capacity to accurately and quickly capture the semantic relationships between words, critical for NLP tasks. Which risks hindering the model's ability to generalise from training to real-world applications effectively. Degrading the overall performance of transformers.
    - Need to find papers to cite for this, else I'm basing off nothing but books and articles.

Benefits of Layer Normalisation (LN)

Henceforth, LN is used, maintaining consistent distributions through (11). This results in several theoretical improvements:

- **Reduces Internal Covariate Shift:** Since feature dimension is normalised, internal covariate shift is reduced. Hence, training cost is reduced, and performance is increased, as model will likely be able to capture semantic relationships more easily. Refer to findings of (XX Research paper).
- **Minimises Hyperparameter Sensitivity**: By standardising activations across layers, LN lessens the dependency on precise initial parameter settings. Facilitating easier and more robust model training, by reducing the need of hyperparameter tuning.
- **Alleviates Vanishing Gradient Issues**: According to (Yao, et al. 2019), vanishing gradient problem is addressed through maintaining a consistent activation scale (distribution). This prevents the risk of dead neurons, and facilitates more effective training.

[7]Disclaimer: This is by no means a comprehensive list. And is also just theoretical, since we don't fully understand what's going on behind neural networks. (will refine this later)

Effects of Layer Normalisation

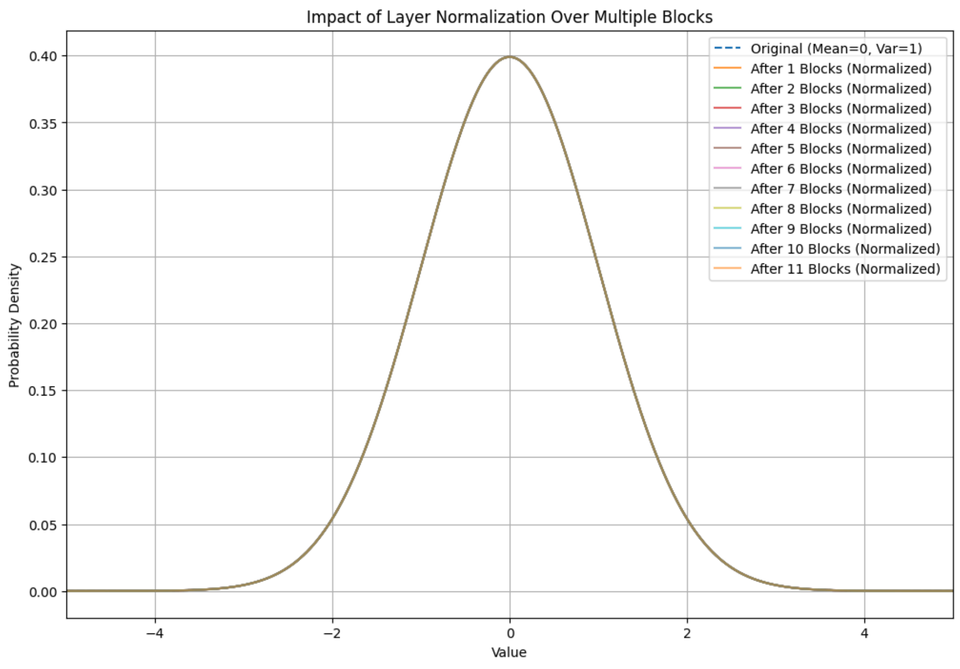


Figure-7: Distribution of input data when Layer Normalisation is applied

- From this, we can see how the input's distribution to each block is always consistent. That is, LayerNormalisation ensures that our input will always has a mean of 1, and variance of 1.

Section 4.2 — Root Mean Square Normalisation

RMSN is a more computationally cheaper counterpart than LN. It's used by novel LLMs such as Meta's LlaMa series, contrasting to LN which is used by OpenAI's GPT series.

RMSN is very similar to LN, with the only difference being these operations:

Effects of Layer Normalisation

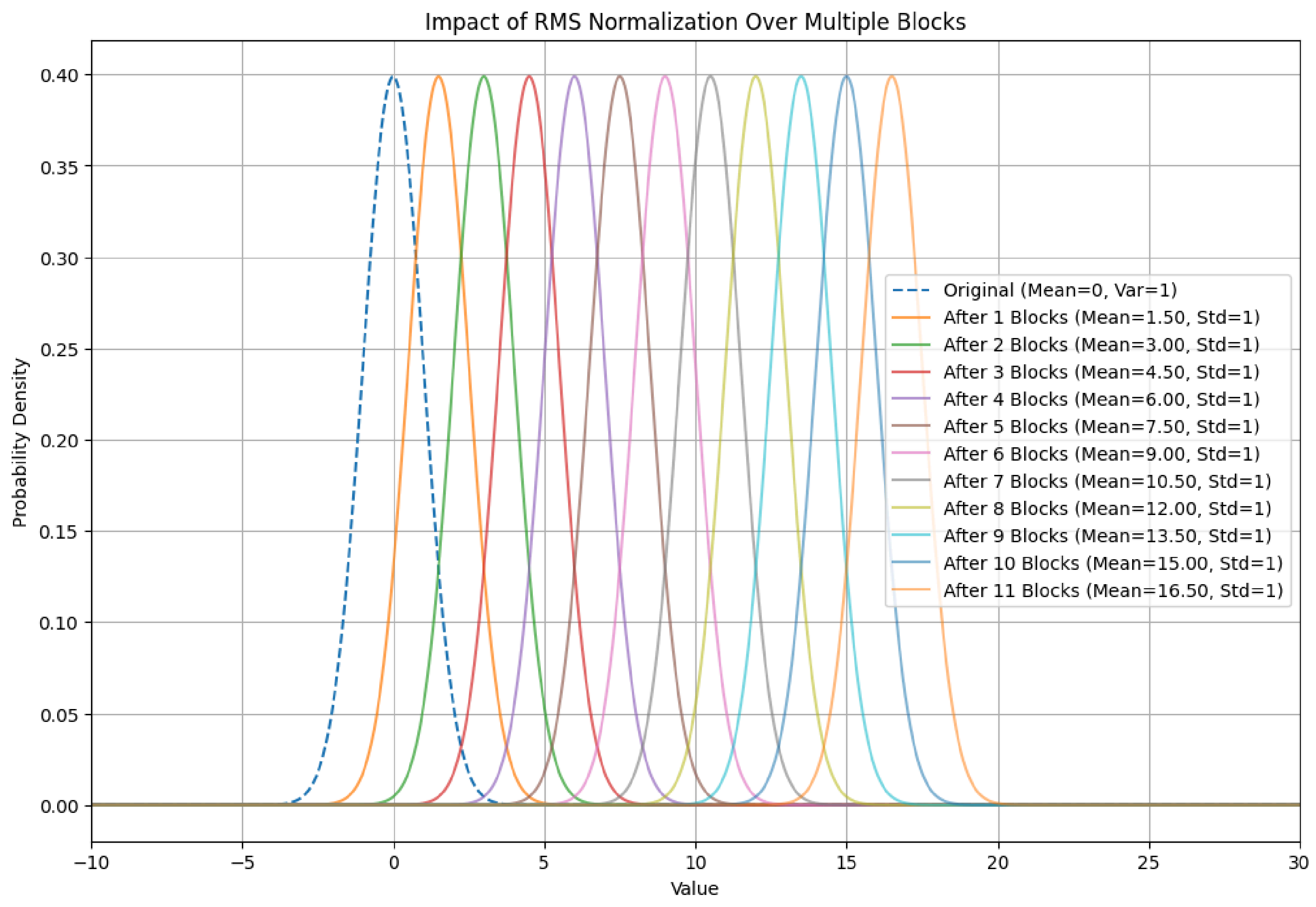


Figure-8: Distribution of input data when Root Mean Square Normalisation is applied

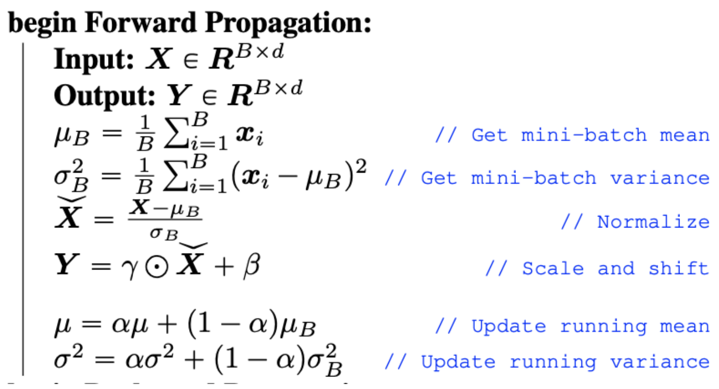
- We can see the effects of not shifting the distribution's mean back to 0 on the other hand. Meaning that the mean of the input distribution will keep shifting rightwards. (Given the assumption that each block causes the input to shift by 1.5, and the input's variance to scale by 1.2x)

Since we are not calculating the mean, and not shifting out input's mean to 1. In theory, RMSN should be quicker than LN to run.

Section 4.3 — Power Normalisation

PN is a novel normalisation method proposed by Sheng-Shen. With its idea being to rethink the way to implement batch normalisation[8] for the transformer architecture.

Implementing batch normalisation naively in transformers was done prior, but had always led to significantly worse performance than Layer Normalisation. Hence, the changes proposed by Sheng-Shen is such:



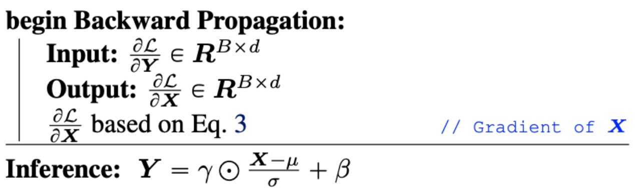
```
begin Forward Propagation:
    Input: $\boldsymbol{X} \in \boldsymbol{R}^{B\times d}$
    Output: $\boldsymbol{Y} \in \boldsymbol{R}^{B\times d}$
    $\mu_B = \frac{1}{B}\sum_{i=1}^{B} \boldsymbol{x}_i$                        // Get mini-batch mean
    $\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(\boldsymbol{x}_i - \mu_B)^2$    // Get mini-batch variance
    $\breve{\boldsymbol{X}} = \frac{\boldsymbol{X}-\mu_B}{\sigma_B}$                                    // Normalize
    $\boldsymbol{Y} = \gamma \odot \breve{\boldsymbol{X}} + \beta$                              // Scale and shift

    $\mu = \alpha\mu + (1-\alpha)\mu_B$                      // Update running mean
    $\sigma^2 = \alpha\sigma^2 + (1-\alpha)\sigma_B^2$          // Update running variance
```

Figure-9:

<u>Calculation of running statistics</u>

<u>Main methodology</u>

- It's normalising across the batch first, and this is done by dividing over the square root of variance, rather than variance itself. With Sheng-Shen's hypothesis being that, we shouldn't normalise it/clamp it down too hard to the extent which the batch only has the variance of 1. According to Sheng-Shen's analysis of the nature of natural language data. (Batch Normalisation is normally used with CNNs, and worked well because of the nature of image data)



```
begin Backward Propagation:
    Input: $\frac{\partial\mathcal{L}}{\partial\boldsymbol{Y}} \in \boldsymbol{R}^{B\times d}$
    Output: $\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}} \in \boldsymbol{R}^{B\times d}$
    $\frac{\partial\mathcal{L}}{\partial\boldsymbol{X}}$ based on Eq. 3                 // Gradient of $\boldsymbol{X}$
Inference: $\boldsymbol{Y} = \gamma \odot \frac{\boldsymbol{X}-\mu}{\sigma} + \beta$
```

Figure-10:

<u>Approximate Backward Propagation</u>
For more detail, please refer to the paper, but this was done because of the nature of using running statistics. As we are unable to retrieve the precise statistics of what running_phi was when the input was passed through. Hence, approximation is needed.

<u>Inference</u>
As seen in Figure-7 as well, once trained, during inference, Powernorm uses the calculated statistics of running_phi during inference.

[8] (a different type of normalisation technique, normalises across the batch dimension)

Effects of Power Normalisation

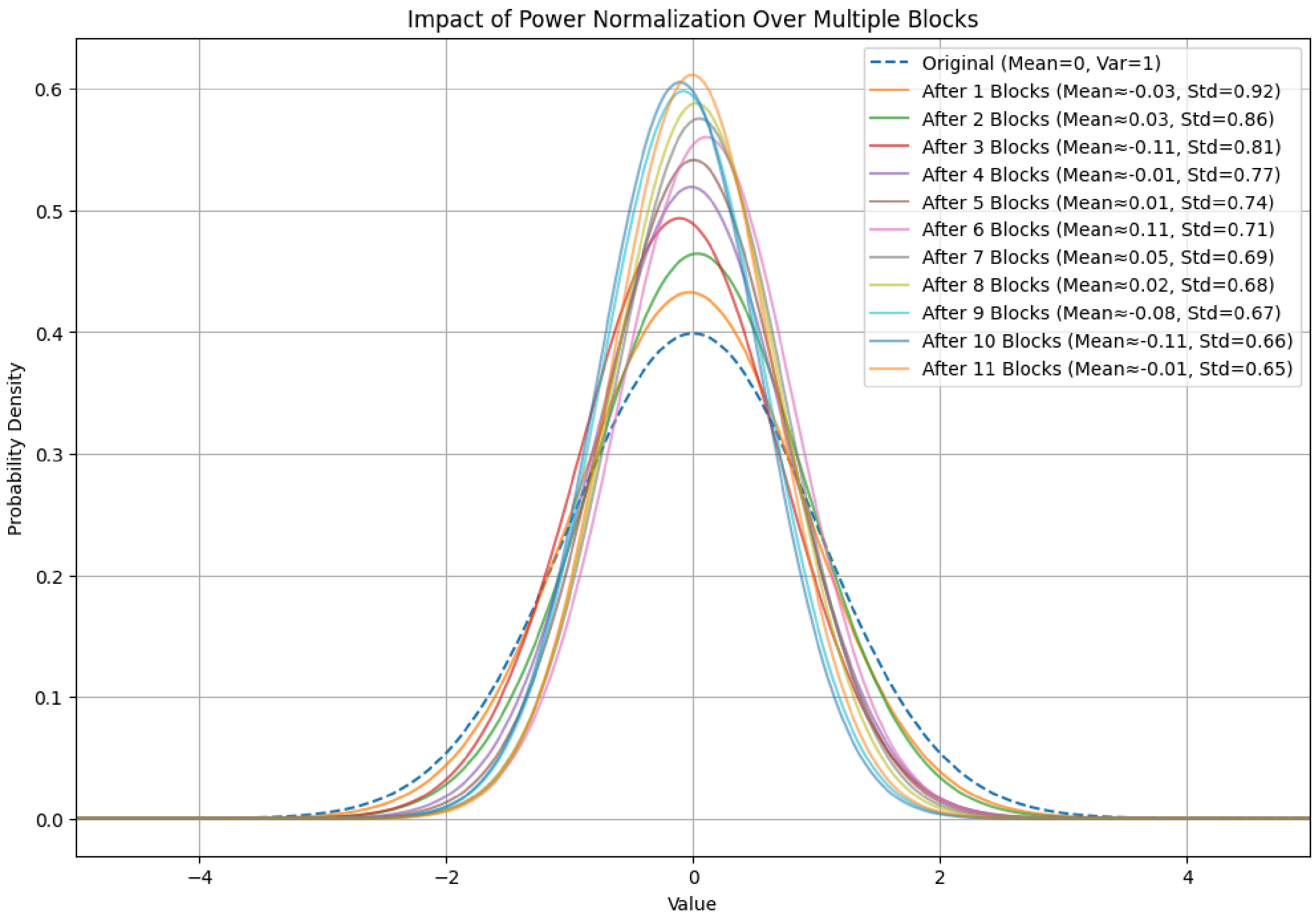


Figure-11: Distribution of input data when Power Normalisation is applied

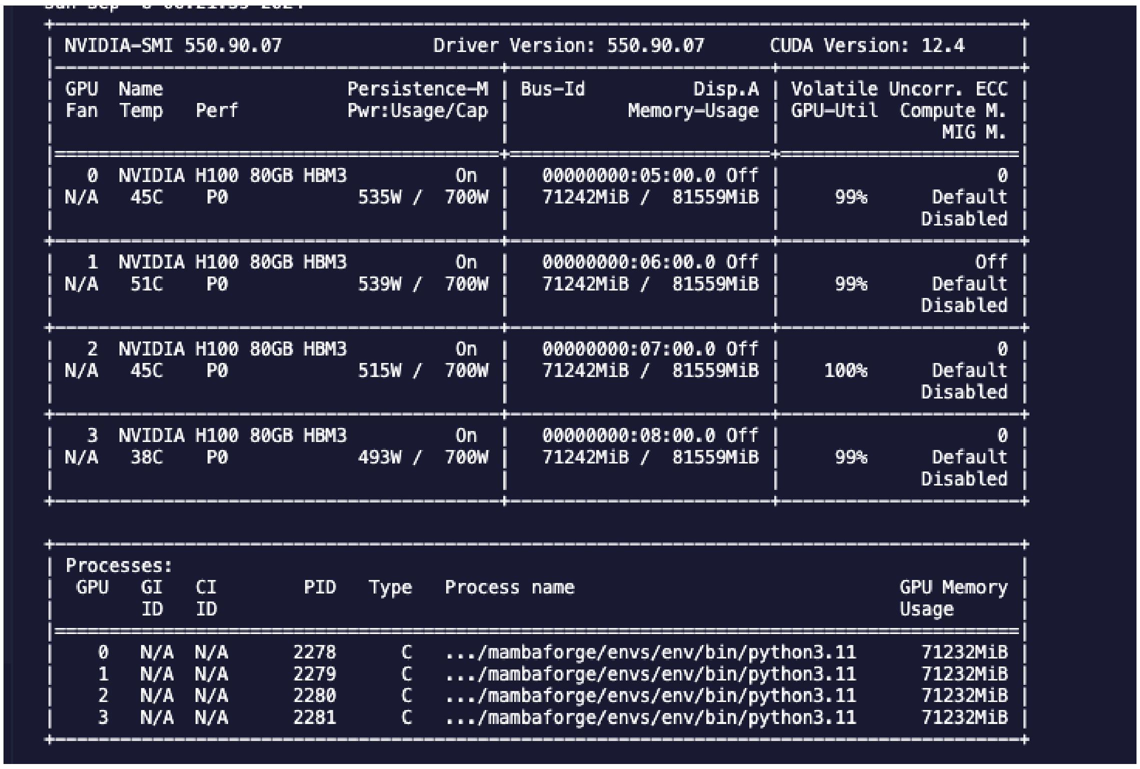
- Since Powernorm is shifting the mean of the distribution by batch statistics, instead of calculating the mean for each layer. This means that the mean of the input distribution will be close to 0, but not exactly 0.
- We can also see the effect of scaling the variance each time by torch.sqrt(varience). As the variance of the curve itself still somewhat increases as the input traverses down the blocks, but is still fairly controlled, compared to not scaling it at all. (This is done based off on Sheng-Shen's theory that scaling by sqrt(variance {running batch statistics}) for natural language)is better than scaling by sqrt(running variance)

Section 5 - Methodology

To be able to test how the different types of LNs fare when pre-training a transformer. Surprise, we need a way of pre-training custom architecture GPTs!

To do this, I had followed through Andrej-Karparthy's "zero-to-hero" course. And used Andrej's nanoGPT's repository's source code. Which the code basically implements the architecture and training loop for GPT-2 in PyTorch, based on the methodology presented in OpenAI's GPT2 paper. The code is then trained on the FineWeb dataset, which entails high quality text corpus scalped by Huggingface.

Section 5.1 — SuperComputer



```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.90.07              Driver Version: 550.90.07      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| 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  NVIDIA H100 80GB HBM3          On  |   00000000:05:00.0 Off |                    0 |
| N/A   45C    P0            535W /  700W |   71242MiB /  81559MiB |     99%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA H100 80GB HBM3          On  |   00000000:06:00.0 Off |                  Off |
| N/A   51C    P0            539W /  700W |   71242MiB /  81559MiB |     99%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   2  NVIDIA H100 80GB HBM3          On  |   00000000:07:00.0 Off |                    0 |
| N/A   45C    P0            515W /  700W |   71242MiB /  81559MiB |    100%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   3  NVIDIA H100 80GB HBM3          On  |   00000000:08:00.0 Off |                    0 |
| N/A   38C    P0            493W /  700W |   71242MiB /  81559MiB |     99%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      2278      C   .../mambaforge/envs/env/bin/python3.11      71232MiB |
|    1   N/A  N/A      2279      C   .../mambaforge/envs/env/bin/python3.11      71232MiB |
|    2   N/A  N/A      2280      C   .../mambaforge/envs/env/bin/python3.11      71232MiB |
|    3   N/A  N/A      2281      C   .../mambaforge/envs/env/bin/python3.11      71232MiB |
+-----------------------------------------------------------------------------------------+
```

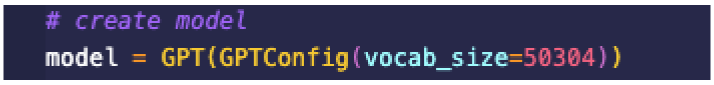
Figure-12: The Nvidia H100 supercomputer which I'm renting for this EE

Training GPT from scratch is a very computational intensive task, even for smaller 124-million parameter and 345-million parameter models! (for reference GPT-4 has 1.8-trillion parameters) Hence, I had rented Nvidia-H100 supercomputers to meet our computational needs! Providing us a whopping 1.2 petaflops.

Note: To save money, I tend to use spot instances, which are much cheaper. For example, the setup in Figure-12 would've normally costed me 11.8 USD. But it only costed 5.413 USD when spot instances are used.

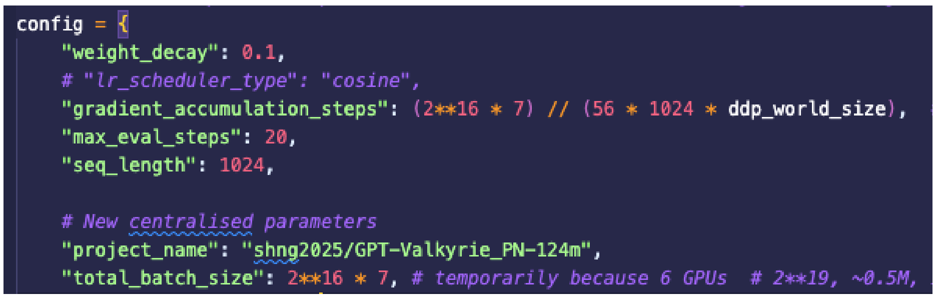
Section 5.2 — Speeding up nanoGPT

- One of the technique is that throughout the architecture and the training loop, its most preferable for us to use numbers with lots of powers of 2s. For example:



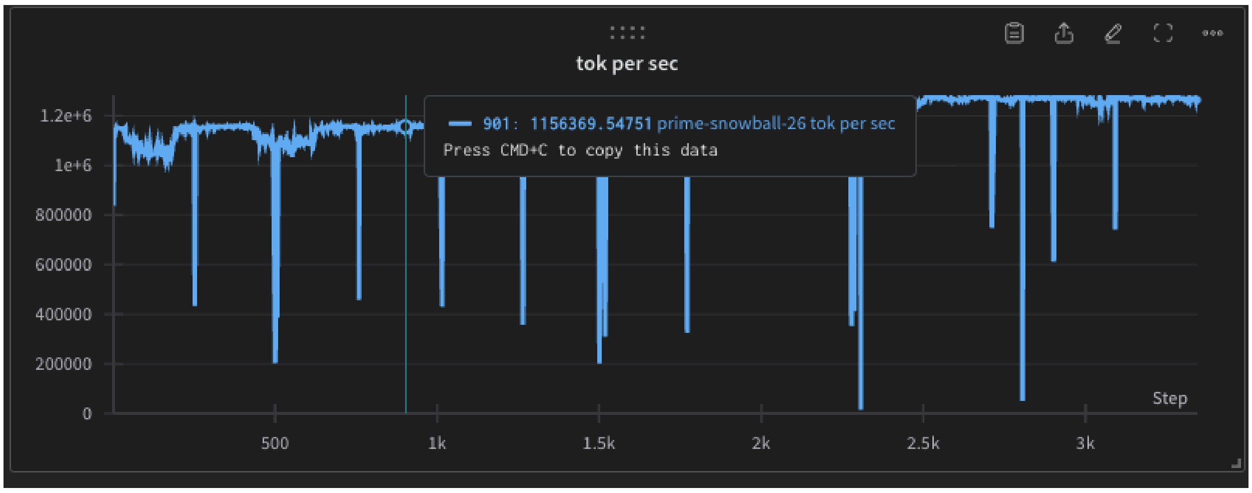
```
# create model
model = GPT(GPTConfig(vocab_size=50304))
```

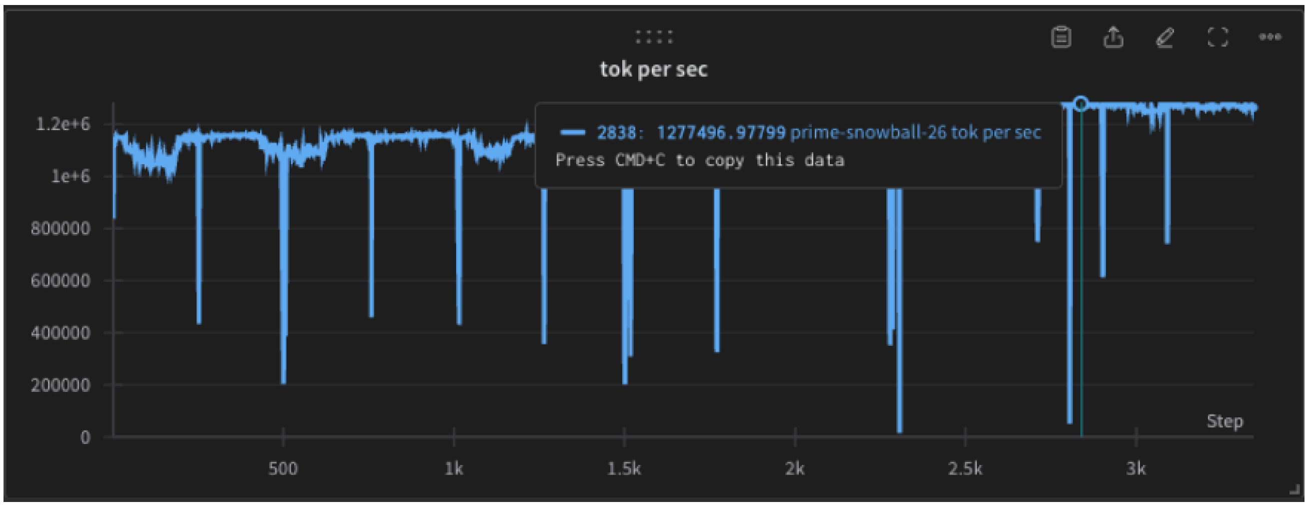
  - Even when we have a token size of 50257 in theory, we used 50304 because it has lots of powers of 2s.



```
config = {
    "weight_decay": 0.1,
    # "lr_scheduler_type": "cosine",
    "gradient_accumulation_steps": (2**16 * 7) // (56 * 1024 * ddp_world_size),
    "max_eval_steps": 20,
    "seq_length": 1024,

    # New centralised parameters
    "project_name": "shng2025/GPT-Valkyrie_PN-124m",
    "total_batch_size": 2**16 * 7, # temporarily because 6 GPUs  # 2**19, ~0.5M,
```

  - Total_batch_size being 2**19 instead of 500,000.
    - This is the configuration recommended by OpenAI. Why? Because its empirically proven to be faster. Its because due to the nature of Nvidia GPUs and how CUDA works. Because CUDA tends to process things in powers of 2s. And when your not using numbers with lots of powers of 2s, you will be underusing the full capabilities of your respective GPUs, which causes slower code.





    - This is also evidence, as we see how 6 GPUs is actually slower than 4 GPUs

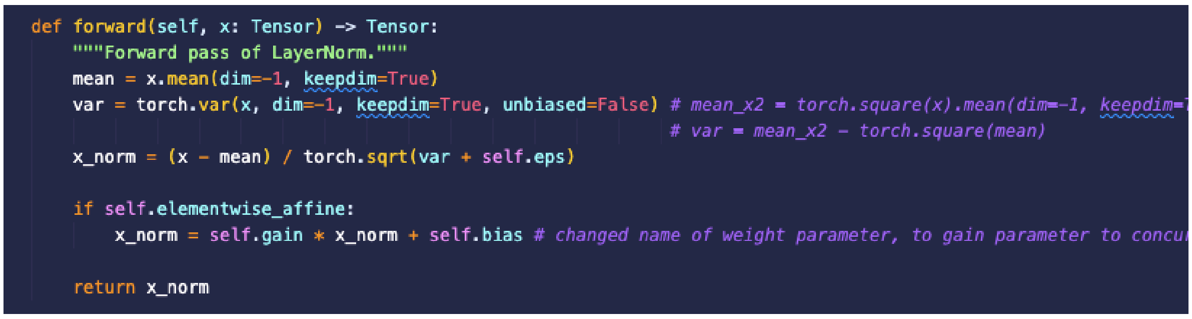
        - Had to change total_batch_size
    - 1 more thing to point out is tokens/sec doesn't scale up linearly with more GPUs. For example, single GPU vs double GPU {**need to be graphed out empirically TDL**}

- Torch.compile is used. Because it basically compiles everything into C/CUDA code. Which based on our empirical testing, had increased tokens/sec by 3 fold.
- TF32 is used. Speed things up further.

- We are using Weights and Biases to monitor our training results.
- I'm using Hugging Face to store my huge models for free.

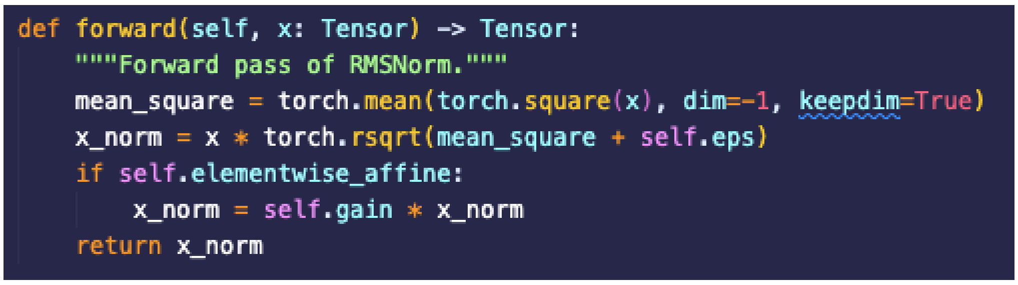
<u>Section 5.3 — Implementation of Custom Architectures</u>

- Layer Normalisation



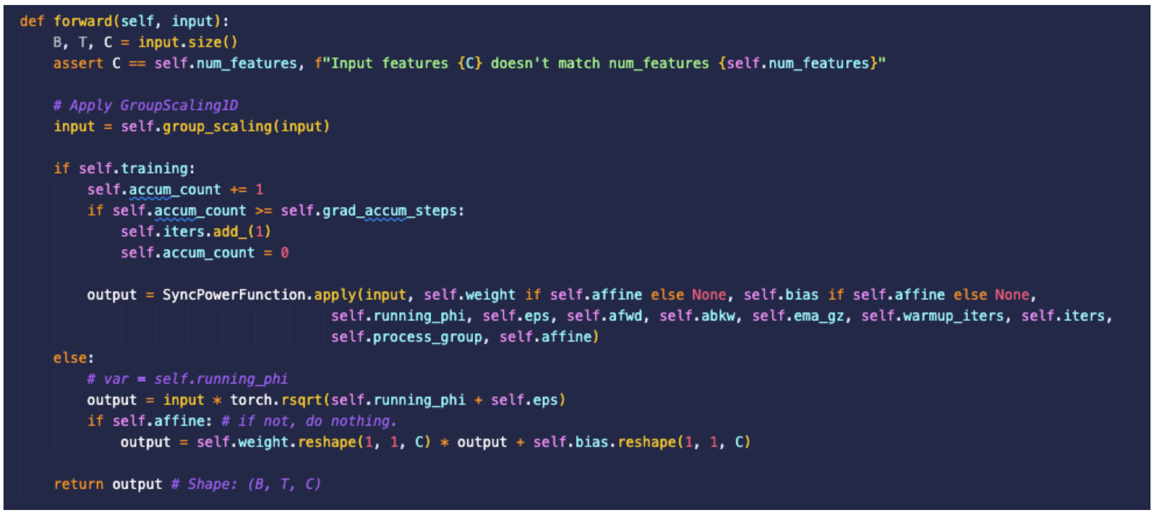
```python
def forward(self, x: Tensor) -> Tensor:
    """Forward pass of LayerNorm."""
    mean = x.mean(dim=-1, keepdim=True)
    var = torch.var(x, dim=-1, keepdim=True, unbiased=False) # mean_x2 = torch.square(x).mean(dim=-1, keepdim=
                                                              # var = mean_x2 - torch.square(mean)
    x_norm = (x - mean) / torch.sqrt(var + self.eps)

    if self.elementwise_affine:
        x_norm = self.gain * x_norm + self.bias # changed name of weight parameter, to gain parameter to concu

    return x_norm
```

- RMS Normalisation



```python
def forward(self, x: Tensor) -> Tensor:
    """Forward pass of RMSNorm."""
    mean_square = torch.mean(torch.square(x), dim=-1, keepdim=True)
    x_norm = x * torch.rsqrt(mean_square + self.eps)
    if self.elementwise_affine:
        x_norm = self.gain * x_norm
    return x_norm
```

- Power Normalisation



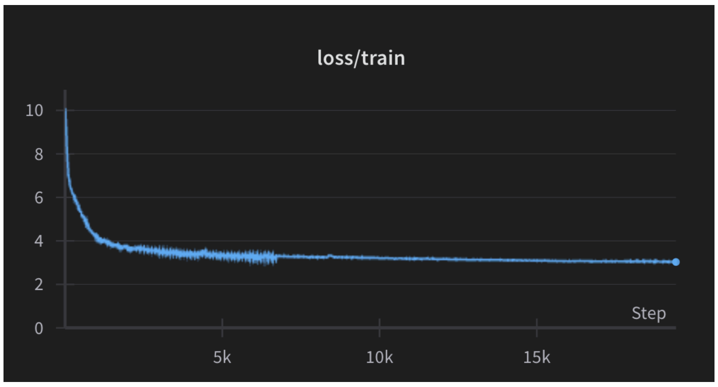
```python
def forward(self, input):
    B, T, C = input.size()
    assert C == self.num_features, f"Input features {C} doesn't match num_features {self.num_features}"

    # Apply GroupScaling1D
    input = self.group_scaling(input)

    if self.training:
        self.accum_count += 1
        if self.accum_count >= self.grad_accum_steps:
            self.iters.add_(1)
            self.accum_count = 0

        output = SyncPowerFunction.apply(input, self.weight if self.affine else None, self.bias if self.affine else None,
                                         self.running_phi, self.eps, self.afwd, self.abkw, self.ema_gz, self.warmup_iters, self.iters,
                                         self.process_group, self.affine)
    else:
        # var = self.running_phi
        output = input * torch.rsqrt(self.running_phi + self.eps)
        if self.affine: # if not, do nothing.
            output = self.weight.reshape(1, 1, C) * output + self.bias.reshape(1, 1, C)

    return output # Shape: (B, T, C)
```

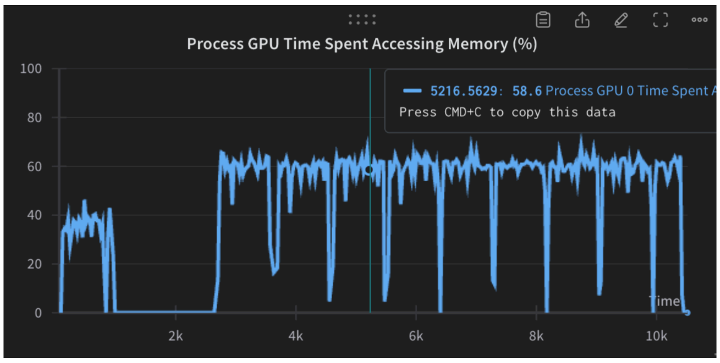
For PowerNormalisation, i had relied on the source code provided by S1ncerass on his github repository.

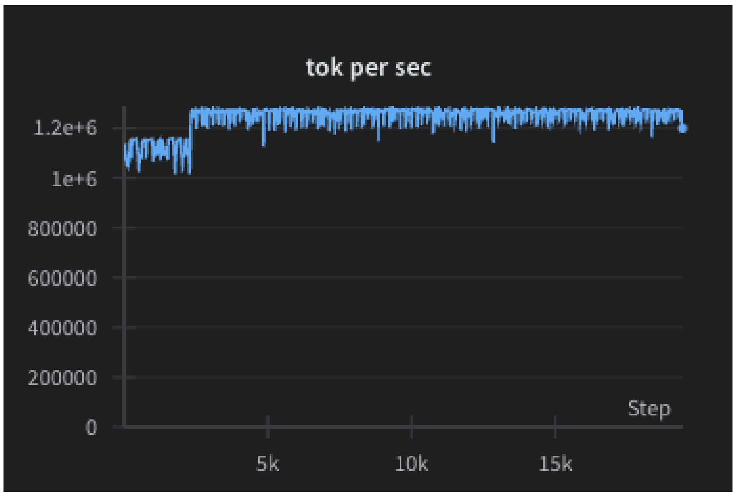
Since his code wasn't implemented for DDP or parallel execution, I made some modifications personally and implemented torch.dist functions such as:

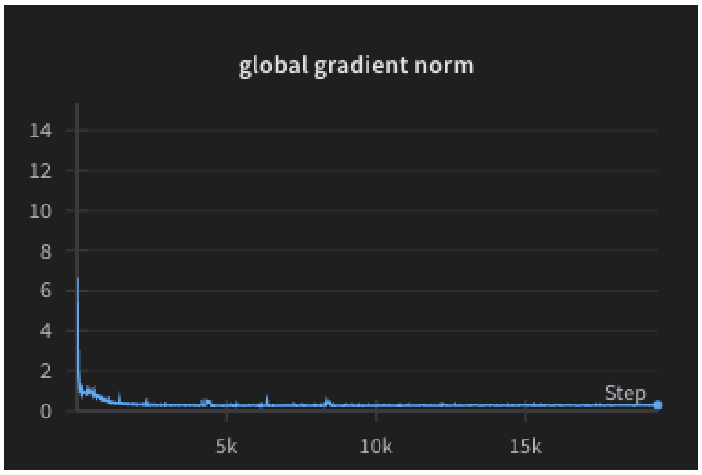
Section 6 - Empirical Results and Data Analysis

Section 6.1 - Layer Normalisation (Training Curve)



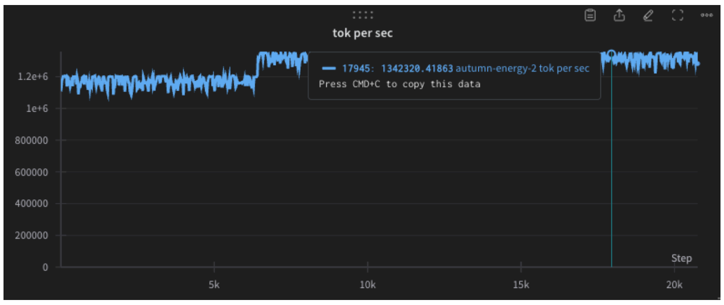


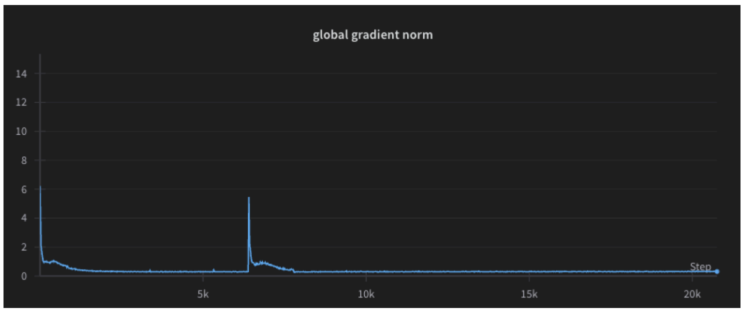


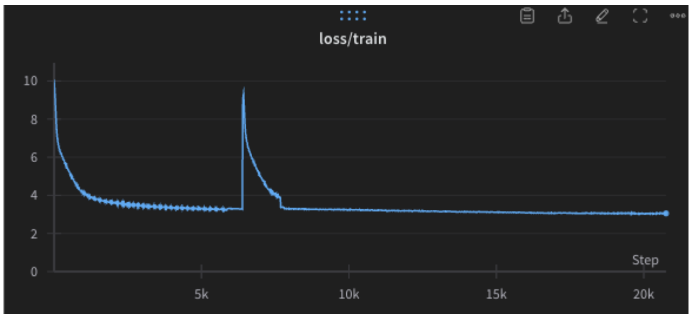


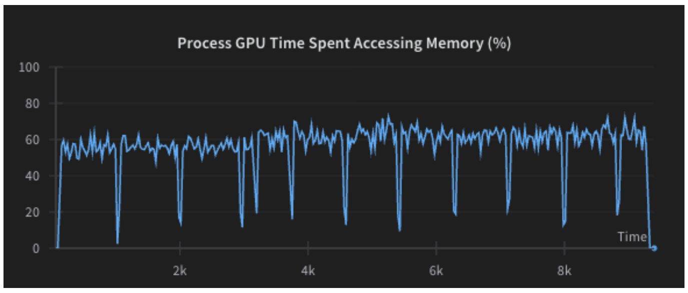
- Loss curve looks very stable. Converged early on and remained at 3.4 loss/train throughout the run.
- Global Gradient Norm is very stable, which indicates that no explosive gradient had occurred which is a very good sign.
- tokens/sec which is at about 1.2 million. We will use this as baseline to compare against RMSN and PN.
- Time spent accessing HBM is at 58%, which is fairly good. (time spent accessing memory needs to be minimised when possible, because its slower based on empirical data. Because HBM is a bottleneck)

Section 6.2 - RMS Normalisation (Training Curve)





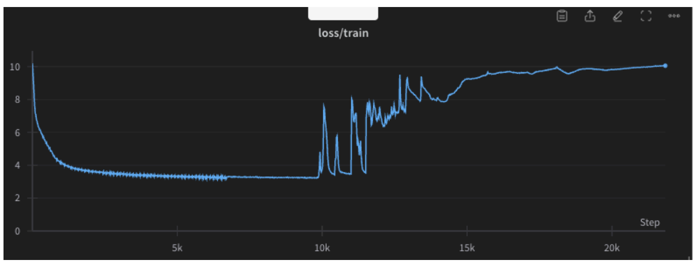


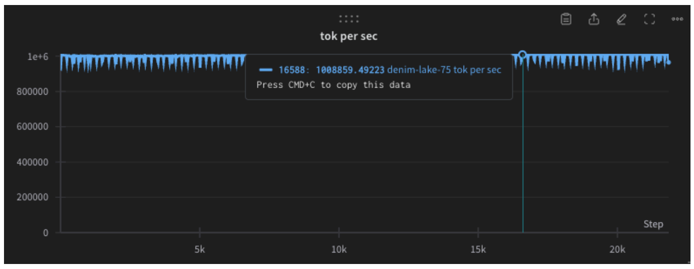


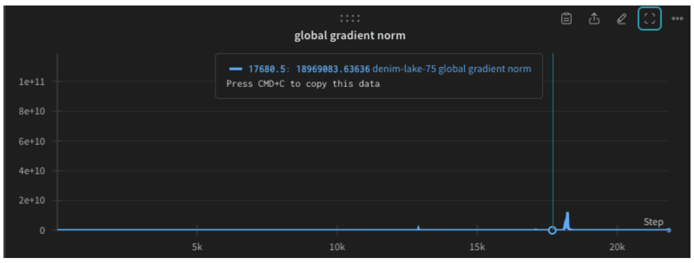
- Loss curve is very stable once again, akin to RMSN. So is global gradient norm indicating no explosive gradients.
- Tokens/sec as hypothesised is actually faster, by 5%.
- Time spent accessing memory is similar.

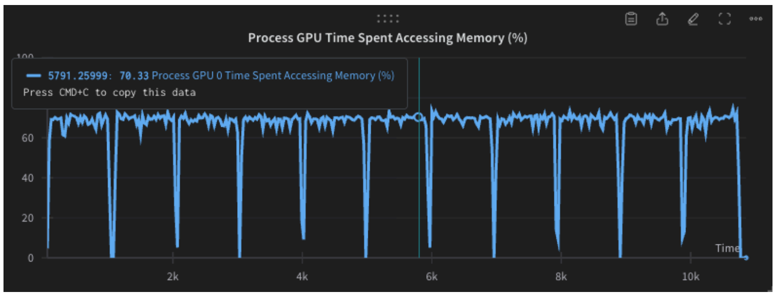
Section 6.3 - Power Normalisation (Training Curve)

- Refer to appendix for further issues faced with Power Normalisation + globalScaling1D + NaN error









- Power normalisation by itself is slower, at 1.08 * 10^6 tokens/sec. 15% slower than layer normalisation. And due to the nature of requiring a larger operation, and using running statistics, a huge part of this slowdown likely stems from the fact that Power Normalisation spends a lot more time accessing memory. That is 70%.

- Up till step 10000. The global gradient norm had rapidly increased, all the way up to 1e10. Meaning the gradient had exploded.

- Run had failed. Meaning the normalisation technique isn't ideal.

Section 6.4 - Running Inference

In this section, we will run inference off our pre-trained model to check if they are appropriately trained.

Section 6.6 - Common Evaluation Metrics

Section 6.5 - Initial Conclusion

Section 7 - Ablation Studies

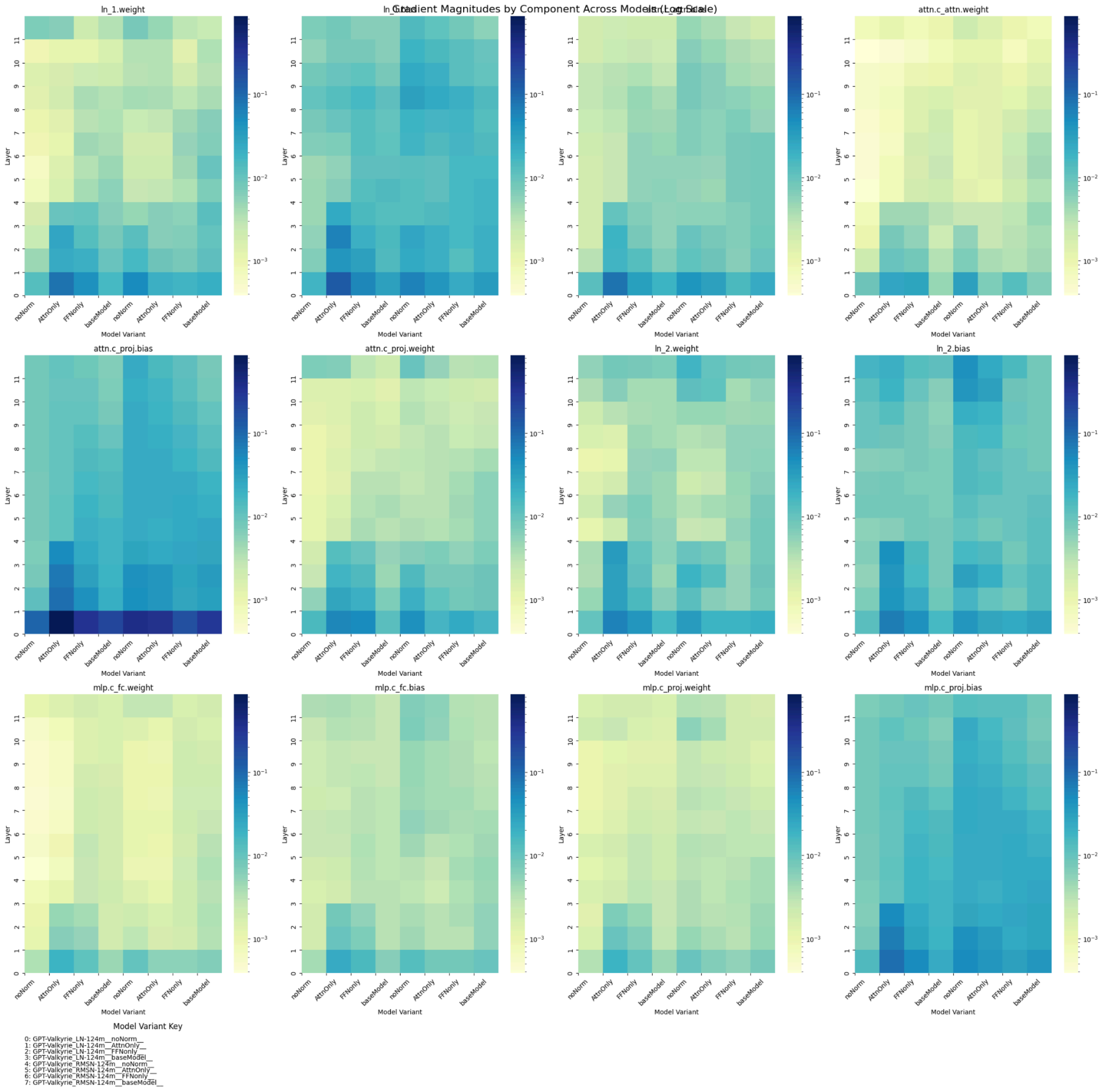
Since we can confirm that power normalisation itself had failed, we will only be running ablation studies from here onwards only on Layer Normalisation and RMS Normalisation.

One of the methods which we will evaluate these models is via Ablation Studies:

- What is ablation studies. Why it can provide us insights.

Section 7.1 - Analysis of ablated models

- Below is the graph showing the ablated models running an exemplar code and we see its loss gradient when we run backpropagation onto it:



Section 8 — Fine-tuning Ablated Models

In order to further study our

Section 9 — Analysis of Fine-tuned Ablated Models

Section 10 — Final Conclusion

Section 11 — Evaluation and Future Directions

Experimental Results and Reflection

(1550 words)
Analysis

- Evaluating metrics
    - BLEU,
    - hellaswag?
    - fine-tuning for a few tasks
- Raw Result Analysis
- Experimental Analysis and Limitations
- Conclusion

(275 words)
Evaluation and Improvements

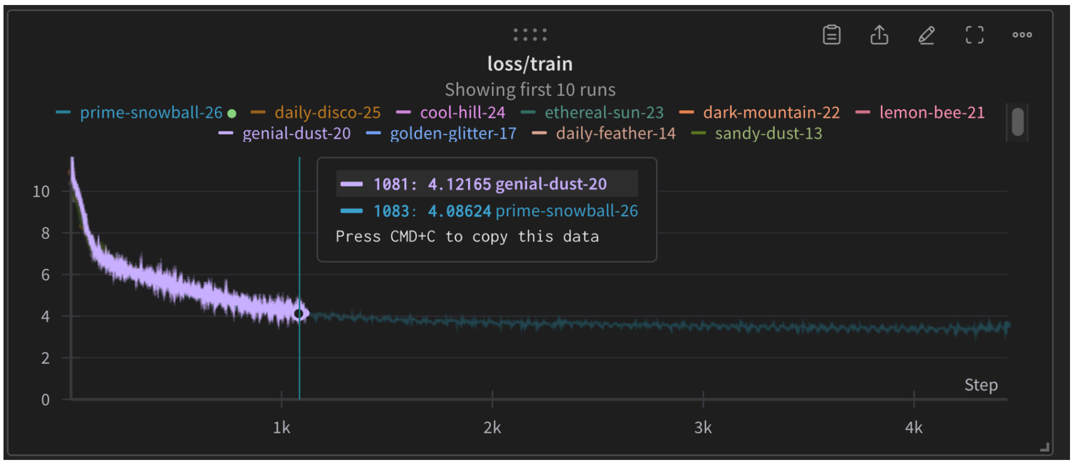
- Further improvements
    - larger models, more variations, larger token context
    - better fine-tuning ?
    - multi-modal ? (a lot more compute will be needed)
- Evaluation
    - better fine-tuning techniques probably will cause more impacts
    - probably futile to be doing little optimisations like this? as better to instead wait for even stronger compute, allowing for newer architectures to be used

Appendix

<u>Originally wanted to implement GPT4 tokenizer instead</u>

Originally I wanted to use GPT4 tokenizer instead of GPT2 tokenizer. Under the impression that it uses a better ReGex hence will lead to better performance. As for coding for example, it recognises tabs as 1 character, rather than 4 separate characters.

But this is proven wrong as seen in this run:



```
1081: 4.12165 genial-dust-20
1083: 4.08624 prime-snowball-26
Press CMD+C to copy this data
```

- It was twice as slow.

Genial-Dust-20 is GPT4 tokenizer--"Cl_100k_base"

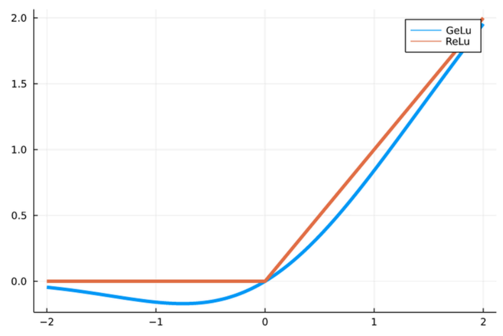
- Single GPU run for both "Genial-Dust-20" (GPT4 tokenizer) and "Sandy-Dust-13" (GPT2 standard tokenizer)
- As seen in image above - GPT4 tokenizer surprisingly doesn't seem to have any improvement over using standard GPT2 tokenizer. When compared to prime-snowball-26 (GPT2 tokenizer, but with 4 GPU) at 1000 steps.
- Lower tokens per second process rate can be seen. Due to a series of potential factors:

- Due to the fact that GPT2 tokenizer only has 50,257, whilst CL_100K_base, have, surprise surprise, 100k+ tokens, actually means we can't use FP16, since it can only represent 2**16 types of different numbers.
- This means that our total_batch_size had to be shrunk from 2**19 to 2**18, else the GPU keeps running out of space.
- And the main slowdown is likely due to how much slower information takes to be transferred, or memory overhead between HBM or CPU-GPU connection.
- GPU also processes slower when using a higher float, even tho we've already tried TF32.

Background Information - Quirks of Deep Learning

In this section, I will explain key concepts utilised by the transformer architecture:

Non-linear Activation Functions (72 words)



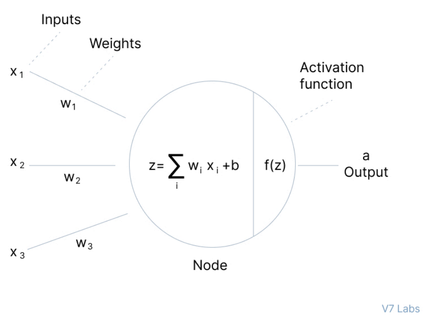


figure 2.1, 2.2

Figure-2.1 displays GELU, an activation function. It determines if specific output values should be activated, thus controlling information flow to subsequent layers (figure-2.2). Non-linear functions like GELU are essential in neural networks. Because if a transformer solely contains linear functions, no matter how many linear functions one has, it can always be represented by a single linear function only. Hence non-linear functions like GELU enable the transformer to capture complex relationships.

Backpropagation and Optimizers (63 words)

The flowchart shows the backpropagation process where the model adjusts its weights based on the error gradient of the output. Accompanying this, a table of different optimizers like SGD and Adam highlights how these algorithms help minimize the loss function. Optimizers vary in how they adjust learning rates and handle gradient descent, directly impacting the convergence speed and stability of the model's training.

Tokenizer and Positional Embedding (52 words)

A

Q, K, V, and Self-Attention (78 words)

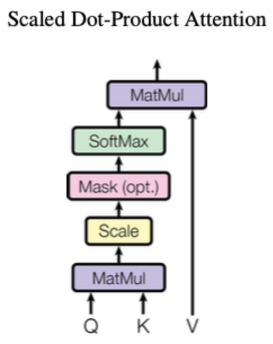


figure 3

$$Attention(Query,\ Key,\ Value)\ =\ softmax(\frac{Query \bullet Key^T}{\sqrt{d_k}})\ \cdot\ Value$$

Query, Key, Value matrices are derived from multiplying input embeddings by learned matrices. The dot products QK^T are scaled by $\sqrt{d_k}$ to ensure numerical stability during learning. Softmax converts these values into a probability distribution, effectively selecting the input parts to focus on.

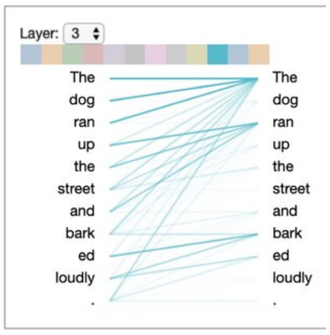


figure-3

This operation basically tells the model to pay attention to more contextually significant words. As seen in Figure-4 higher weights from $softmax(\frac{QK^T}{d_k})$ on the word "the" help the model emphasise *the*'s $Value$ when processing.

<u>*Feedforward Networks (54 words)*</u>

The block diagram outlines the structure of the feedforward layers in a transformer. Post-attention, the processed data passes through these layers, which apply further transformations to refine the model's output. Each layer is fully connected and operates on the principle of transforming input features into higher-level representations before passing them to the next layer.

<u>*Layer Normalisation (56 words)*</u> The diagram illustrates layer normalization which standardizes the inputs across the features within a layer. By adjusting and scaling the inputs, layer normalization helps in stabilizing the neural network's training. It is crucial for combating the internal covariate shift, ensuring that each layer receives data within a scale that prevents the vanishing or exploding gradient problem.

→ MORE DETAIL GUIDE ABOUT

1. *RESIDUAL CONNECTIONS,*
2. *TEMPERATURE OF ATTENTION,*
3. *MULTI-HEAD ATTENTION*
    a. *WILL BE ADDED IN APPENDIX. SINCE THEY ARE IRRELEVANT COMPARED TO WHAT WE ARE EXPLORING.*
4. *MORE DETAIL ON SOFTMAX(),*
5. *ACTIVATION FUNCTIONS (GELU vs RELU), ← why non-linear is needed*
6. *Backpropagation and optimisers (intro to ML ← why minimum gradients are needed)*