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

| (100 words)  **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)  **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:    *“The brown fox jumps over the lazy dog”*  Currently, we have only prompted our transformer with:    “*The brown fox jumps”*  And given prompt , I will show you how does the transformer generate text and completes the sentence to achieve ! |
| *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 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:    *Note: This is an oversimplification, as we are only showing a .* |
| *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:        Figure-2: Graphing out Positional Embedding  Which, we can visualise more clearly how the positional embedding applies for our prompt :          Hence, from , we can see an example of how positional embedding generates unique positional embeddings for each text in a given position! As seen in Figure-2, 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 is then added, so that our token contains both the semantic meaning and positional information of words in :    Our prompt:      Great! Now that our input contains both semantic meaning and positional information, we can start processing it to predict the next word! |
| *Q, K, V, and Attention (78 words)*    *figure 3*    First, we need to pass our embeddings into the “Multi-Head attention” which contains multiple “Self-Attention mechanism”:  How does self-attention work?   1. matrices are derived from multiplying to learn matrices.    1. : It's like a searchlight pointing at something we're interested in.    2. : It's like labels on items we're searching through.    3. : Holds the content of each item. 2. : This operation determines how relevant each item’s label is to our searchlight . 3. is scaled by ( in our case) to ensure numerical stability during learning.[[1]](#footnote-0) 4. normalises these value to , effectively selecting the input parts to focus on. 5. Multiplying by : Forms a weighted sum, here based on the relevance of each items—, their respective content are emphasised.     *Figure-3: Visualisation of GPT-2’s self-attention given the input “The brown fox jumps” using BertViz[[2]](#footnote-1)*  Basically, this operation tells the model to pay attention to more contextually significant words. As seen in Figure-3, given our prompt , for all words, more “*attention*” is placed on the word “”.  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”:    Figure-4: Multi-head Attention formula from “Attention is All you need” paper[[3]](#footnote-2)  Which, let’s visualise what happens with our :    From , you can hence see how the first row, which represents the word retains a heavier weighting than the other 3 rows, which represents . Now, our model will pay more attention to during processing, since it has a heavier weight! |
| *Layer Normalisation (56 words)*  Layer normalisation, as seen in Figure-1, this normalisation operation is applied on our input after operations such as “Multi-head attention” and “Feedforward” to reduce internal covariate shift. Here’s how it works:    *Note: Layer normalisation is applied for every ith row, hence represents each row. For our exemplar below, we will only use*  *\_for simplicity. We are also going into more detail here, since Layer Normalisation is the focus of our research.*  **How does it work?**   | Operations of | Sample Calculation | | --- | --- | | 1. mean of is calculated with: | *Since we are using \_only:* | | 2. the variance is calculated with: |  | | 3. : This operation shifts to have a mean of 0. | *Note: Calculating ’s mean now results in 0.* | | 4. : Our input (with mean now equal 0), is divided by its sample standard deviation . This scales down (or up) the values in each row in our input to have of 1. | *Note: Calculating ’s now results in 1.* | | 5. But, normally, (a small value) is added to the scaling function .  So that we don’t risk dividing dividing by in the event when approaches . This avoids instability during training. | *Why is*  useful? *Assuming in the event which our approaches :*  *- Without* :     * 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 (for GPT-2, )*:   * As seen here, the exploding gradients error has been subverted. Preventing risk of training run failure. |   Internal Covariate Shift  The reason we shift to and to , is to address the issue of internal covariate shift. This issue occurs as the distribution of the [[4]](#footnote-3) 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 is processed by a *[[5]](#footnote-4)* without , its is shifted by , and its is scaled by . And that will be passed through a transformer with 12 [[6]](#footnote-5).    Figure-4: Distribution of after passing through *N*x blocks.  As seen in Figure-4, assuming our originally has and . We can see how rapidly the distribution gets out of hand! After passing through *11*x , our —the input of the *12*th —distribution had gone completely out of hand! Now having and !  Why is Internal Covariate Shift bad?   1. **Increased Training Time**: Each layer has to continuously adapt to new data distributions because the parameters (weights and biases) optimised for one data distribution may not be effective for another. This constant need to adapt extends the amount of time required for the network to converge on effective parameters, increasing training cost. 2. **Degradation of Layer Effectiveness**: Each subsequent layer relies on the outputs from previous layers, as seen in Figure-4, the impacts of each block accumulates. If early layers continually shift their outputs drastically, later layers are effectively always trying to hit a moving target. This can degrade the effectiveness of deeper layers, which may never receive the kind of stable input necessary to perform their intended transformations reliably.   *Feedforward Networks (54 words)*  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. |
| (175 words)  Background Information - Layer Normalisation   * brief history on CNN + RNNs + BN (very brief only words) |
| (1000 words)  Theoretical Background - LN   * what does LN do (maths stuff begins)   + im expanding upon the work of BLN, but applying PN instead * RMSnorm vs PN vs LN vs BLN/PLN (m)   + explain their differences   + show architecture, differences   >> Formulates hypothesis here, based on differences, etc. |
|  |
| (500 words)  Experimental methodology and materials   * GPUs → Ubuntu, Nvidia A100, 96x-AMD-CPUs * PyTorch, implemented GPT-2 based on Karparthy’s course   + my modifications, using GPT-4 tokenizer instead. why? (150 words)   + torch.compile explanation ← speeding up (50 words?) ++ FLASH attention, CUDA speed up   + gradient checkpointing, HF   + Datasets → fineweb.edu     - upload to HF   + model sizes (n-embd) 128m, 753m, 1.5B   + sequence length – 1024 to 4096 * Implemented layer normalisation by code, and using sIncerass’ code, and facebook’s RMS code, and personal implementation of BLN (substituting) |
| 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 |
| --- |
| **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*  Query, Key, Value matrices are derived from multiplying input embeddings by learned matrices. The dot products are scaled by ​​ 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 on the word “the” help the model emphasise *the*’s when processing.  *Feedforward Networks (54 words)*  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.  *Layer Normalisation (56 words)*  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. |
|  |
|  |

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 [↑](#footnote-ref-0)
2. (jessevig) [↑](#footnote-ref-1)
3. (Google Inc #) [↑](#footnote-ref-2)
4. This refers to the data that passes through each layer of the transformer, like the tensor *(10).* [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
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.* [↑](#footnote-ref-5)