Analyzing the Effects of Targeted Structural Pruning of Attention Heads in Autoregressive LLMs

Jack Settles
Institute for Artificial Intelligence
University of Georgia
Athens, Georgia
John.Settles@uga.edu

***Abstract*—Large Language Models (LLMs) have proven to be incredibly useful tools for automating and improving tasks involving natural language. However, their main drawback is their size, which has implications for both memory and latency concerns. This paper explores the use of structured pruning to reduce model size and improve latency during inference. The structures we target for pruning are the attention heads inside a transformer LLM's self-attention mechanism. As we will come to see, this technique provides little reward.**

***Index Terms*—structured pruning, LLM, attention heads, layerwise-relevance propagation, LRP**

I. Introduction

Since the introduction of OpenAI's ChatGPT to the public in November of 2022, the term "Large Language Models" (LLMs) has become commonplace in nearly every domain, from machine learning, law, business, medicine, and everything else in between. The "Large" in this term is certainly not without merit; the sheer size of these models is directly related to their power and impact. It is also directly related to their power *consumption*, along with their enormous memory requirements, and their latency (i.e. the time it takes to process inputs and outputs). In order to be effective tools, LLMs must be able to process and generate natural language in real time for users. This process involves computing a series of matrix multiplications, which is slow on serial devices like CPUs. As a result, GPUs are a necessity due to their parallel processing capabilities.

The benefits of fast inference on GPUs do not come without costs, the most notable of which is the reduced memory capacity on these devices. A high-end server grade GPU such as an Nvidia A100 has a maximum storage capacity of 80GB of VRAM. Commercial grade LLMs far exceed this amount, requiring the model to be hosted on a server with multiple GPUs, with the model partitioned across those devices. This means that these models are only useful tools in areas with internet access to reach the models' API endpoints. A highly congested network could slow down processing times, negatively impacting user experience. Then there are also privacy concerns associated with mandating users' data to be sent to these servers.

All of these aforementioned concerns have prompted researchers to explore the possibility of building smaller language models that can be deployed on local machines. Model compression is one way to achieve this result, with the goal of reducing the memory footprint of a model. This is typically done through techniques like quantization [1], distillation [2], or pruning [3], [4]. Quantization involves modifying the data type of the weights in a model such that their overall memory footprint is reduced. Most models are trained and stored in (floating point) FP32 or FP16 format. Reducing a model's weights from FP16 to Int4, for example, would mean that each weight goes from being represented by a 16 bit floating point number down to a 4 bit integer, resulting in a 75% reduction in memory for the weights alone.

Distillation involves compressing the latent knowledge of a larger neural network into a smaller one. This is typically done by taking a larger pre-trained model to serve as the "teacher" to a smaller "student" model. During training, the goal of the student model is to align its output predictions with those of the teacher. In short, the teacher gives the student hints to the answers, and the student benefits from the learned representations from the teacher.

Model pruning involves severing the connections between neurons in a neural network. This method rests on the assumption that not all of the weights in a neural network are needed for the task at hand, so they therefore can be removed.

In this paper, we explore the effects of structured pruning in an effort to reduce the memory footprint of modern autoregressive LLMs. The models we utilize for these experiments are Llama-3.2-1B-Instruct and Llama-3.1-8B, with 1 billion and 8 billion learned parameters, respectively. The structures we aim to prune are the attention heads inside the model layers' self-attention mechanisms. To identify which heads to prune, we utilize layer-wise relevance propagation (LRP) [5].

The remainder of this paper will follow as such: section II will cover previous literature on neural network pruning, along with an overview of LRP; section III will discuss our implementation methods and testing protocols; section IV reviews our results; and sections V-VII will conclude and address some limitations and potential further research.

II. Background

A. Pruning in Artificial Neural Networks

First proposed in [3] under the term "Optimal Brain Damage", neural network pruning involves removing weights in a network in order to reduce the size of the model, while still maintaining a high accuracy. This methodology rests on the assumption that not all weights are necessary for the model's objective. Neural networks are typically implemented as a series of matrix transformations, with the matrix multiplication operation being the most fundamental type of transformation. In a matrix multiplication, the rows of one matrix and the columns of another are used to compute dot-products. A dot-product consists of multiplying the values element-wise in each row/column vector before summing them up. So, if any of the values in either vector are at (or close to) a value of 0, then the resulting multiplication will be 0, or close to it. The end result is that these values contribute very little to the output value, and are therefore not needed. The authors in [3] refer to this "close to 0" heuristic for pruning as "saliency". They also note that it is merely one of many ways to identify which weights to prune.

Within model pruning, there are a few different approaches. Reference [6] actually considers model pruning to be a spectrum, where the two ends are dubbed fine-grained and coarse-grained pruning.

Fine-grained pruning (a.k.a. *unstructured pruning*) involves removing individual weights (i.e. individual scalar values) from the model's weight matrices. For instance, following the saliency heuristic of [3], all weights that are close to a value of 0 can be removed with little to no effect on the model's accuracy. Given the enormity of modern deep neural networks (DNNs), it is almost inevitable that some of the weights will be close to 0. This is especially the case if a penalty like L2 regularization is added to the model's loss function during training. This added penalty would push many smaller weights closer to 0.

The major drawback to fine-grained pruning is that it tends to incur some overhead. This overhead comes in the form of both extra storage requirements and extra computational steps. The extra storage requirements come from the fact that, with a fine-grain pruned weight matrix, the indices for each retained weight must now be tracked explicitly. In practice, this does not tend to consume a lot of memory: usually the reduction in number of parameters sufficiently outweighs the increase in memory consumption that comes with storing the retained weight's indices. This extra storage only tends to increase memory usage when the pruning rate is so small that storing the indices for all of the retained weights results in *more* memory usage than what was saved by removing the pruned weights.

While explicitly storing the row and column indices for retained weights does not typically increase memory consumption, it does incur extra *computational* overhead during the forward pass for the model, enough such that real-time performance is often affected. Because each row and column index must be stored for each retained weight, we can no longer utilize dense matrix multiplication operators like GEMM. For a dense matrix (where all values are retained), their location in memory is implicit, since the matrix values are stored in contiguous blocks of memory. All one needs to perform a dense matrix multiplication is the memory address for the start of the matrix. The rest of the indices (and their corresponding memory addresses) can be obtained with the shape of the matrix. Because of this structure, memory access is predictable, so modern hardware, such as GPUs, is highly optimized for performing these operations very quickly. In contrast, unstructured pruning introduces a high degree of irregularity in this process. The values that are retained cannot be known at the outset by the compiler. Although indexing into an array has a complexity of *O(1)*, doing this for each individual weight is still much slower than dense matrix multiplications. Due to this extra lookup step for each weight, inference latency tends to increase with fine-grained pruning.

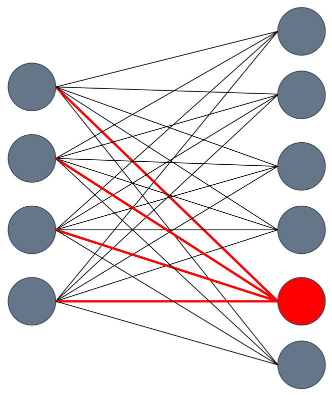

Fig. 1: An example of structural pruning, where all weights that are tied to an output neuron are removed, effectively removing that neuron entirely. Unstructured pruning would be akin to removing only some of the weights connected to an output neuron. That neuron would remain, but be comprised of fewer terms in its weighted sum.

Opposite to fine-grained/unstructured pruning is coarse-grained/structured pruning, as seen in Fig. 1. Knowing what we now know about unstructured pruning, the pros and cons of structured pruning can easily beinferred. Structured pruning involves removing entire structures in the weight matrices of a neural network, such as entire rows or columns. Removing entire rows or columns means that the "denseness" of a matrix can be retained by simply packing the retained rows/columns into smaller dense matrices. Since the pruned matrix is still a dense one, we can still take advantage of its implicit structure and use optimized matrix multiplication kernels. So, with coarse-grained pruning, latency is often retained if not improved. The major drawback to structured pruning, though, is that removing entire contiguous elements of a weight matrix leads to severe accuracy drops for the model. Even if nearly all weights in a row or column are close to 0, if only one of them has a high enough magnitude to affect the output value, then removing that entire row or column will lead to a very different output, resulting in a drop in accuracy.

B. Layerwise-Relevance Propagation

Layer-wise Relevance Propagation (LRP) was first proposed in [5] as a method for machine learning explainability. Given that deep neural networks are largely black boxes, it can be difficult to discern why or how a model came to a decision for its final prediction. LRP attempts to trace a "relevance" score for a given prediction back through the model to the input features. This allows one to identify which input features are most important for a prediction. Initially, LRP was used to identify which pixels in an input image are most relevant to making a prediction on image classification tasks. For instance, if a model correctly classifies that an image is of a dog, ideally it is basing this prediction on the pixels that correspond to the actual dog, and not other superfluous features like the background, or perhaps a watermark that coincides with all of the dog images. This can be used to verify whether a model is making valid predictions, or if it is relying on coincidental correlations between features and classes.

LRP begins by initializing a relevance vector. This vector has the same shape as the output logits from the model; it is full of all zeros, except for the index with the highest value logit. This index serves as the prediction, so the raw logit value serves as the initial relevance score for that sample's prediction. A high-level overview of LRP is that it attempts to disperse this logit value back throughout the network layer by layer. For a purely linear transformation, the relevance of a neuron is equal to the proportion in which it contributes to the weighted sum of the neurons in the next layer, which is just its input activation times the weight connecting it to the neuron in the next layer. This is then multiplied by the relevance at the next layer for that next layer neuron.

Beginning at the output layer with the relevance vector initialized as all zeros except for the predicted value, the relevance for each of the previous layer's neurons can be computed following Eq. (1). z_{ij} is equal to the activation coming into neuron i times the weight connecting neuron i (in layer I) to neuron j (in layer J). This process is simple to compute for the final layer because we are only concerned with one output j neuron; however, once that output neuron's logit value has been dispersed to the previous layer, the relevance vector is much less sparse. When propagating to the layer before that, we now have to consider how much each previous neuron contributed to all of the neurons in the next layer that they connect to. Once again, this is merely the proportion that each neuron contributes to the weighted sum of each of the next layer's neurons, times the relevance for each of the next layer's neurons.

$$R_i = \sum_j \frac{z_{ij}}{\sum_i z_{ij}} R_j \tag{1}$$

In theory, LRP seems quite simple. In practice, however, it runs into issues due to the fact that most DNNs are not purely linear transformations. Modern DNNs consist of linear transformations, plus non-linear activation functions. Transformers make use of multi-head self-attention as well, which includes a softmax operation. These architectural features pose challenges to reliable LRP attributions because, from one layer to the next, the total amount of relevance should be preserved. Also, if relevance values become very small, then continuously multiplying these values by the previous layer neurons' proportions could lead to a numerical underflow issue. The same can be said if the relevance of a neuron is rather large, with an exploding relevance value. This is no different from the vanishing/exploding gradient problem that is well known in literature on DNN training.

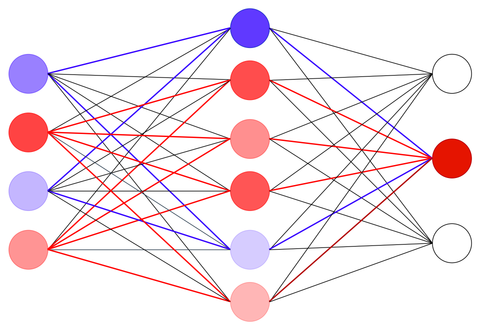
Fig. 2: Simple LRP example for a small feedforward neural network. The prediction is the middle neuron in the last layer, with red lines connecting to neurons in previous layers with a high relevance. Blue lines and neurons represent neurons with a low relevance score.

To circumvent these issues of numerical instability, a number of additional LRP rules have been proposed [5], [7], [8]. Some of these rules treat neurons with positive incoming activations differently from negative ones, and some work [8] proposes using a composition of multiple rules for a model, with the specific rule depending on each layer. It is worth noting that most of these rules were developed for convolutional neural network architectures, so their effectiveness for transformers is unknown.

In this paper, we attempted to implement each of these rules by hand for a transformer-based language model, but nonetheless we ran into the well-reported issues of numerical instability. Luckily, recent research [9] has been devoted to adapting LRP to transformer architectures, for both language modeling and vision transformers. The methodology for computing the relevance of a neuron in [9] follows a slightly different approach compared to previous work in order to get around the non-linearities in transformers. Leveraging the automatic differentiation capabilities in PyTorch [10], these authors use a gradient*input method for calculating relevances for neurons. Backpropagation is typically utilized to get the gradient of a loss function with respect to each weight in a network, which is then used to update the weights accordingly to minimize the overall loss. Instead of taking the gradient of some loss function, the authors simply take the gradient starting from the model's output prediction with respect to each neuron along the way. At a high level, these gradients tell us how much each neuron contributed to the computation of the final predicted logit. Multiplying these gradients times their neuron's input activations gives us the relevance for that neuron for that particular prediction. Although LRP was initially developed to assign a relevance score to input features, it can be utilized to assign relevance to intermediate activation values. This is due to the fact that the input features' relevances depend on the first intermediate layer's relevances, which in turn depend on the next intermediate layer's relevances, and so on.

III. Implementation

In this project we utilize the framework developed by [9] to calculate relevance scores for intermediate activations of two open-source LLMs. Specifically, we calculate the relevance scores for each attention head in each layer of the models. We then use these aggregated relevance scores over many samples to determine which attention heads can be pruned. The hope with this methodology is to find some attention heads that, across all kinds of different input sequences, are low-hanging fruits. If some attention heads provide very little to the output predictions regardless of the content in the input sequence, then we can conclude that those heads are largely redundant and not necessary.

A. Models

The two open-source models we use are Llama-3.2-1B-Instruct and Llama-3.1-8B [11], both from Huggingface. These models were chosen primarily for 3 reasons. First, as they are open source, they can be downloaded, saved, and modified locally, which is crucial for obtaining latent relevance attributions for the hidden states. This is also necessary for pruning/modifying the weight matrices of the model, which cannot be done for proprietary models. Second, these models are some of the newest releases, representing a portion of the state-of-the-art in modern LLMs. Finally, these models are small enough to fit inside the memory of a single A100 GPU, making this research easy to implement. The relatively small size of them also meant we could iterate quickly, scaling up the amount of data for both relevance attribution and potential fine-tuning, while also running our benchmark tests quickly.

It should be noted that, since the second generation of Llama [12], these models utilize grouped-query attention (GQA) [13]. With the original multi-head attention implementation, each token embedding is projected into h query, key and vector embeddings, where h is the number of heads. So for a model that uses 32 attention heads per layer, each token will have 32 queries, 32 keys, and 32 values. Another way to put it is that a sentence goes from 1 representation of a sequence to 32 different smaller representations of the sequence. Attention scores are then computed within each head with the respective queries, keys and values.

With GQA (as implemented in Llama 3), on the other hand, each token embedding gets projected to 32 Q embeddings, one for each head. For K and V, however, each token is projected into only 8 representations. The head dimensions for all of these QKV projections is still the same so that attention can be computed between them. With GQA, every 4 Q projections corresponds to 1 K and 1 V projection.

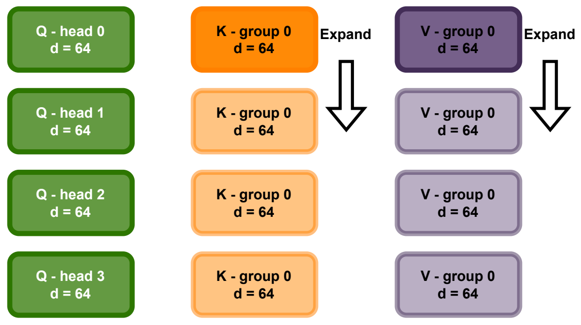

Fig. 3: Grouped-Query Attention as implemented in Llama 3. Each token embedding in a sequence is multiplied by each QKV matrix. Q matrices output a vector that is the same length as the input vectors (embeddings). K and V output vectors that are $1/4^{th}$ the length of their input. The Q output is split into 32 distinct heads. The K and V outputs are split into 8 distinct group vectors. During the forward pass, these K and V embeddings get expanded to match the number of heads in that group. The 4 green Q embeddings are unique, but the 4 orange K embeddings are the same, just duplicated. Same goes for the 4 purple V embeddings.

B. Hardware

All tests were run on the University of Georgia research cluster Sapelo2, provided by the Georgia Advanced Computing Resource Center (GACRC). We used both Nvidia L4 and A100 GPUs, with the type and number depending on the test.

C. Data

The dataset used for this project comes from the BabyLM challenge [14]. The BabyLM challenge aims to build small language models on a cognitively plausible amount of data, with the largest dataset consisting of around 95M words. This dataset is constructed from subsets of 6 different corpora spanning everything from child directed speech to Wikipedia articles. This dataset was chosen for its diversity of language scenarios, and its quick availability.

D. Profiling and Pruning Procedures

First, each model had to be profiled to determine which attention heads could serve as potential targets for pruning. For each of the subcorpora in the BabyLM dataset, we randomly sampled 5000 sentences from it. These can be anything from short/aphoristic phrases to longer/more complex sentence constructions. Since we are working with language models, relevance propagation calculates the relevance of the next token prediction for the entire input sequence. This means that if we only calculated relevance scores for attention heads based on full sentences as inputs, then the relevance would always be with respect to predicting what comes after the end of a sentence. So, the relevance would always be with respect to predicting what comes after, say, a period, or a question mark, or some other end of sentence punctuation. Since this does not provide a very holistic overview of the model's learned linguistic knowledge, we passed in sub-sequences to the model. Specifically, we passed in every possible sub-sequence for every sample sentence. This way, the obtained aggregated relevance scores would be with respect to the model having to predict the first token, and then the second, then the third, and so on. By doing so, the model's relevance scores reflect its predictions for all possible words, reflecting the differences in the distribution of types of words in a language.

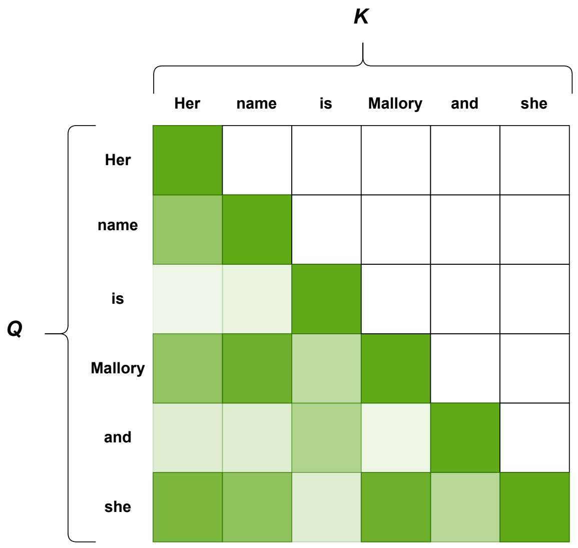

Fig. 4: A single square attention matrix. For this attention head, attention scores for *Her*, *name*, *Mallory* and *she* are relatively high, indicating that this head may be responsible for coreference resolution, i.e. matching words that correspond or highly correlate to the same entity.

Each attention head represents intermediate activation values for the model during the forward pass. Specifically, a given layer's attention heads are represented as a 4-dimensional tensor in the shape of [batch_size, number_of_heads, sequence_length, sequence_length] (we use a batch size of 1). In a square attention matrix for autoregressive language modeling, an upper-right-triangular mask is applied. This means that the first row corresponds to the first token's query vector, the second row to the second token's query vector, and so on. Each column represents each token's key vector.

In order to obtain a singular relevance value for each head, we must aggregate all of the attention head relevances together. We could simply take the average along the key dimension, and then the average along the query dimension. However, averaging along the key dimension can be misleading. Since each query can only attend to its own key and the keys before it, the average relevance across a row is not the sum divided by the length of the row. Rather, the average relevance for a row would be its sum divided by *the number of tokens that query was able to attend to*. We call this a masked-mean aggregation. Once this is computed along the key dimension, we then take the standard average along the query dimensions. This average represents the aggregated relevance score for a head in a given layer for a given input sequence. This is done for all heads in all layers for all input (sub-)sequences.

After profiling the model, heads with relatively low relevance scores were selected as pruning candidates. Since the values in an attention head are computed using matrix multiplication between that head's q and k^T projections, pruning the entire head would amount to pruning the rows and columns in the attention module's weight matrices that correspond to those q and k projections. The same can be said for that module's v and output projections: the dimensions of those matrices that correspond to a candidate head for pruning are removed. To implement this, all one needs to know are the heads to keep, and inherent to that, the heads to prune. For the heads we keep, we simply grab their existing rows or columns in the original weight matrices and pack them into smaller weight matrices, effectively discarding the pruned heads' rows/columns.

After adjusting the weight matrices for an attention module by removing the particular rows/columns, the module's forward method must be patched over. For GQA, every 4 queries is multiplied by 1 key and 1 value. In practice, however, these keys and values are actually expanded (replicated) to match the number of queries for that group. This way all of the matrix multiplications between the qkv projections can be done in parallel. If we remove one head in a group, then instead of having 4 query projections, we only have 3. Therefore, the key and value projections for that group must be expanded from 1 to 3, rather than from 1 to 4. The expansion factor for each k and v projection is determined by the number of heads pruned for that group.

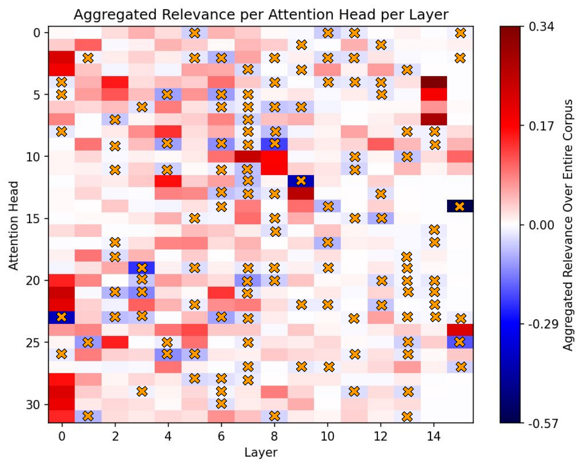

Fig. 5: Relevance heatmap for Llama-3.2-1B-Instruct with 16 layers, 32 attention heads, and a head dimension of 64. The heads we pruned are marked with an ✖. Color scale is set by the min and max relevance values found across the entire model.

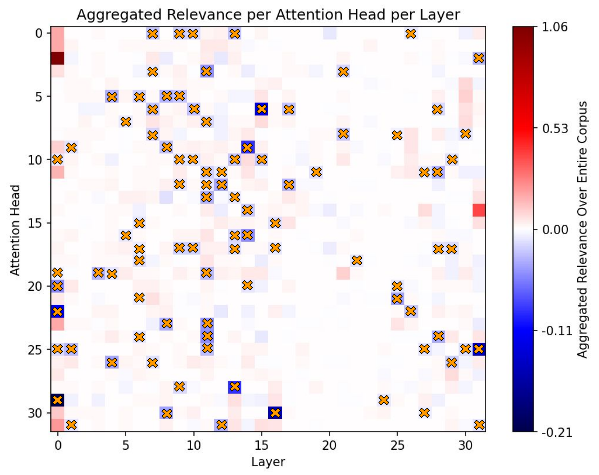

Fig. 6: Relevance heatmap for Llama-3.1-8B with 32 layers, 32 attention heads, and a head dimension of 128. The heads we pruned are marked with an ✖. Color scale is set by the min and max relevance values found across the entire model.

E. Evaluation Procedures

There are four main ways that we evaluate our models. The first is simply comparing the size of the models before and after pruning. In this context, the size is based on the number of parameters in the model. We utilize the thop library for this [15].

The second evaluation is the Benchmark of Linguistic Minimal Pairs (BLiMP) [16]. For BLiMP, a minimal pair is simply a pair of sentences that differ by only one word or phrase. That one word or phrase difference turns a grammatical (English) sentence into an ungrammatical sentence. The goal is to have the model assign a higher cumulative log-probability to the grammatical sequence than the ungrammatical. BLiMP contains 67 tests of 1,000 minimal pairs each. Each test covers a specific linguistic phenomenon (a particular type of sentence construction), and these 67 tests can be grouped into 1 of 12 broader categories.

The third test we evaluate our models on is latency, specifically on pure inference latency. For inference latency, we timed how long it took to perform the forward pass on an input sequence of 64 tokens. We averaged these times over 100 iterations.

The final test is throughput as measured in tokens generated per second (tok/s). For each model, we tested 8 different configurations. 4 configurations utilize caching the prior attention scores, while 4 do not. For each of the 4 caching/no-cache tests, we tested throughput with a short input sequence to generate both 64 and 256 output tokens. We also tested a long input sequence generating again 64 and 256 output tokens.

IV. Results

A. Model Size

Pruning entire attention heads amounts to removing entire rows or columns within the attention module's matrices for a given layer in a transformer. To obtain the 32 query projections that correspond to each attention head, typically each token embedding is multiplied by a singular dense matrix. In the case of Llama-3.2-1B-Instruct, the output of the query matrix is a vector of length 2048. Since there are 32 attention heads, this vector is split into 32 sub-vectors of length 64. So, if we were to remove the last attention head from this layer, then we would remove the last 64 columns of the Q weight matrix.

In theory this sounds like a substantial reduction, especially if we remove multiple heads for a layer. In practice, however, 64 column vectors of length 2048 is only a small portion of the total parameters in the model's attention module. The reality is that when we remove a single head from an attention module, we only reduce the size of the Q matrix by about 3%. The attention parameters alone only make up about 13.6% of the total parameters for the 1B model, and 16.7% for the 8B model. Nothing happens to the K and V matrices unless we remove all of the heads in a single group. We initially considered exploring the effects of pruning at different rates by incrementally increasing the number of pruned heads. However, the current results reflect the maximum number of heads we would consider pruning; given the minimal impact on model size reduction, and the accuracy and latency dropoffs reported in the following sections, we decided that pruning fewer heads would lead to little benefit. The model sizes before and after pruning can be seen in Table I

Model	**Params**	**Params After Pruning**	**% Reduction**
Llama-3.2-1B-Instruct	1.236B	1.202B	2.75%
Llama-3.1-8B	8.030B	7.932B	1.22%

TABLE I: Model size in terms of number of parameters before and after pruning attention heads.

B. BLiMP Accuracies

Our initial inspiration to prune attention heads stems from the fact that LLMs learn patterns across language, and these patterns can be observed in the attention heads [17]–[19]. This makes sense given that multi-head attention is where tokens get to "attend" to each other and effectively learn how they are related. Since syntax imposes formal restrictions on where words can occur in language, we find it plausible that syntax and other syntax-related patterns might be encoded in these attention heads. The authors of [19] actually performed a very similar set of tests as we do in this paper - namely using LRP to identify heads that can then be pruned - but for a smaller transformer on a language translation task. They also did not report metrics such as model size reduction in terms of number of parameters, latency, or throughput.

Despite only pruning 127 out of 512 total attention heads in the 1B model, and 94 out of 1024 attention heads in the 8B model, model performances on BLiMP dropped fairly significantly. Accuracies can be seen in table II. The fact that some of the BLiMP tests decreased so sharply after removing some number of attention heads further supports the belief that some of these syntactic constructions are encoded in these attention heads. If one wanted to look for the specific location of where these phenomena are encoded, they could simply use that BLiMP test as input data for LRP.

Due to time and resource constraints, we were only able to fine-tune the pruned 1B model on a subset of the BabyLM data of about 2M sentences containing a total of 19.5M tokens. The initial validation perplexity for this model was incredibly high, with a value of 113,729, indicating the disruptive effects of attention head pruning. However, after fine-tuning only the attention parameters for 1 epoch, the validation perplexity quickly came back down to 3.04 before leveling off at 3.02 for the next few epochs. Therefore, we cut off fine-tuning after 4 epochs and re-evaluated this model on BLiMP. Despite having a low perplexity value, the model still struggled to capture the grammatical differences in the BLiMP minimal pairs, with its accuracy dropping from 70% to 67%. The results were even worse if we fine-tuned *all* the parameters: the validation perplexity quickly dropped to 1.98 after 1 epoch, but the BLiMP accuracy dropped even further to 59%. This highlights the limitations of perplexity as an evaluation metric. It is also worth considering what pruning does from a mathematical perspective. We typically remove weights that have a low magnitude (or relevance in our case). Although these weights may not have a strong numerical effect on the output, by removing them, we effectively change the function that is our model. We kept the important weights, but they are now related to one another differently than before. A better method would be to follow [6] where the authors fine-tuned and pruned at the same time. This method has the advantage of allowing the model to learn to rely on the non-pruned parameters while the values of the pruned parameters are nudged toward 0.

	1B Original	**1B Pruned**	**8B Original**	**8B Pruned**
Avg. Accuracy	79.59%	70.85%	79.82%	60.40%

TABLE II: Avg. BLiMP scores by model before and after pruning. Individual test scores are not reported due to space concerns.

C. Latency and Throughput

Although pruning attention heads leads to little improvement as far as reducing the size of the model goes, the bulk of the computation in a transformer occurs in the self-attention modules. So, one might presume that reducing the number of parameters in these modules would help improve the latency. When we pruned entire heads, we took the parameters corresponding to the heads we wanted to keep and packed them into a similar but smaller weight matrix. Although this matrix is smaller, it still represents a dense weight matrix, much like it was prior to pruning. Given that modern hardware is extremely efficient at computing dense matrix multiplications, the difference between a Q weight matrix with a shape of [2048, 2048] and a pruned one with a shape of [2048, 1728] after removing 5 heads, for example, has little effect on the latency. As mentioned previously, no changes were made to the K or V matrices unless all 4 heads in a group were removed, which we only did once - the 6^{th} group in the 15^{th} layer of the 1B model. So, the limited reduction in total parameters for the attention modules did not lead to any kind of speedup in latency.

In fact, latency actually got worse after pruning attention heads. As mentioned at the end of Sec. III-C, because these Llama models use GQA, the key and value projections must be expanded so that each heads' queries can be multiplied by the keys and values in parallel. Before pruning, this expansion is simple and constant - all 8 of the key and value projections for each token are duplicated 3 more times, resulting in 4 identical key and value projections for each token for a given group. After pruning, however, we no longer have the same number of heads per group as before: some groups may have retained all 4 of their heads, while others may only have 1 or 2 remaining. This variation means that the repeat factor for each key and value in a given layer will be different depending on how many heads remain for that group (which also varies from layer to layer). As a result, expanding the key and value tensors after pruning takes more time than it does prior to pruning due to having to iterate over the number of repeat factors for each group. Even using native PyTorch functions, such as *repeat_interleave*, that are built on lower level languages like C++ does not offer more benefits than simply iterating over the repeat factors with a for loop.

Throughput is directly related to latency. If the time to compute a forward pass is slower after pruning than it was prior to pruning, then by definition, the number of tokens a model can generate per second will be less than what it can do prior to pruning as well. This is exactly what we observed: the pruned models have a lower throughput than the original non-pruned models.

Test	**1B Original**	**1B Pruned**	**8B Original**	**8B Pruned**
Inference (ms)	15.64	17.36	30.78	33.86
TP-short-64 - cache	64.16	58.35	32.58	29.78
TP-short-256 - cache	64.10	58.35	32.52	29.75
TP-long-64 - cache	64.08	58.37	32.50	29.75
TP-long-256 - cache	63.98	55.33	32.56	29.73
TP-short-64 - no cache	62.58	57.41	32.29	29.40
TP-short-256 - no cache	62.47	56.30	31.36	29.13
TP-long-64 - no cache	62.27	57.20	32.15	29.23
TP-long-256 - no cache	62.42	56.32	29.22	28.11

TABLE III: Latency (ms) and throughput (TP - tok/s) scores for models. Values are averages over 100 runs on an A100 GPU. Short/long describes the input sequence, 64/256 = number of generated output tokens, cache/no-cache = whether we cached the previously generated attention scores.

V. Conclusion

In this report, we looked at the effect of structural attention head pruning on a modern autoregressive transformer LLM. Using two smaller variants of Llama 3 that utilize grouped-query attention, we found that pruning attention heads has a negative impact on the model in a number of ways. First, while it inherently reduces model size in terms of number of parameters, it only does so by a marginal amount.

Second, the average accuracy on linguistic evaluations drops by a significant margin. Although we could have done more evaluations than just testing the model's learned knowledge of language, our thinking is that if an LLM is going to be good at anything, it should be language itself. If the accuracy results after pruning remained on par with the pre-pruned accuracies, then we could have argued that attention head pruning is a quick and easy way to reduce the number of parameters in an LLM, even if only by a small margin. This could have proven beneficial for users with limited storage space, where any reduction in model size might help. However, the reality is that attention heads are crucial for a model's understanding of language.

Third, because these models utilize GQA, the limited benefits from attention head pruning are not even fully realized. Under GQA, attention head pruning only tends to reduce the parameters in the Q matrix. Only when the heads of an entire group are removed do the K and V matrices become smaller. Furthermore, pruning a varying number of heads per group introduces extra computational overhead during the forward pass, slowing down the latency and reducing the model's throughput.

VI. Limitations

This project would not be complete without addressing its limitations. First, simply taking the average relevance scores for attention heads may not be the best way to aggregate and identify candidates for pruning. Since these scores were obtained over numerous sequences and sub-sequences, these aggregated scores tended to balance each other out. Some heads that look extremely irrelevant in Figs. 5 and 6 may not actually be irrelevant, but only less relevant than the others.

As mentioned previously, we were also unable to adequately fine-tune the pruned models. Given the small dataset size we used, it still remains open whether larger-scale fine-tuning after pruning can help bring the model's accuracy back up to its pre-pruned levels. We also only tested this technique on two Llama 3 models, which both use GQA. Our experiments highlight that GQA based LLMs do not benefit from attention head pruning, but it remains open whether other attention variants could benefit from this technique. These Llama models are also relatively small compared to the truly large LLMs that contain tens of billions to hundreds of billions of parameters. These smaller models may already be as optimized and lightweight as they can be, and therefore pruning may not achieve much benefit.

VII. Future Work

While the results of this work were not the most promising, there is still further research to be done. Looking at the aforementioned limitations, potential future work could focus on fine-tuning the pruned models. This could be done after pruning, or pruning could happen iteratively as we fine-tune, as is done in [6]. More interesting research would be to conduct the same experiments in this paper on a much larger model, say a Llama 70B or 405B model. These models are massive, and are quite good at interpreting and producing natural language. With parameter counts that high, we suspect there is some set of structures that are not necessary for language comprehension, and therefore these larger models may benefit more from head pruning.

References

[1] R. M. Gray and D. L. Neuhoff, ”Quantization,” in IEEE Transactions on Information Theory, vol. 44, no. 6, pp. 2325-2383, Oct. 1998.

[2] Hinton, Geoffrey & Vinyals, Oriol & Dean, Jeff. (2015). Distilling the Knowledge in a Neural Network.

[3] Lecun, Yann & Denker, John & Solla, Sara. (1989). Optimal Brain Damage. Advances in Neural Information Processing Systems. 2. 598-605.

[4] Han, Song & Pool, Jeff & Tran, John & Dally, William. (2015). Learning both Weights and Connections for Efficient Neural Networks.

[5] Lapuschkin, Sebastian & Binder, Alexander & Montavon, Grégoire & Klauschen, Frederick & Müller, Klaus-Robert & Samek, Wojciech. (2015). On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. PLoS ONE. 10. e0130140. 10.1371/journal.pone.0130140.

[6] Niu, Wei & Ma, Xiaolong & Lin, Sheng & Wang, Shihao & Qian, Xuehai & Lin, Xue & Wang, Yanzhi & Ren, Bin. (2019). PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning. 10.48550/arXiv.2001.00138.

[7] Binder, Alexander & Montavon, Grégoire & Lapuschkin, Sebastian & Müller, Klaus-Robert & Samek, Wojciech. (2016). Layer-Wise Relevance Propagation for Neural Networks with Local Renormalization Layers.

[8] Montavon, Grégoire & Binder, Alexander & Lapuschkin, Sebastian & Samek, Wojciech & Müller, Klaus-Robert. (2019). Layer-Wise Relevance Propagation: An Overview.

[9] R. Achtibat, S. M. V. Hatefi, M. Dreyer, A. Jain, T. Wiegand, S. Lapuschkin, and W. Samek, “AttnLRP: Attention-Aware Layer-Wise Relevance Propagation for Transformers,” in Proc. 41st Int. Conf. Mach. Learn. (ICML), vol. 235, PMLR, pp. 135–168, Jul. 21–27, 2024.

[10] Paszke, Adam & Gross, Sam & Massa, Francisco & Lerer, Adam & Bradbury, James & Chanan, Gregory & Killeen, Trevor & Lin, Zeming & Gimelshein, Natalia & Antiga, Luca & Desmaison, Alban & Köpf, Andreas & Yang, Edward & DeVito, Zach & Raison, Martin & Tejani, Alykhan & Chilamkurthy, Sasank & Steiner, Benoit & Fang, Lu & Chintala, Soumith. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library.

[11] A. Dubey et al., “The Llama 3 Herd of Models,” arXiv preprint arXiv:2407.21783, 2024.

[12] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, et al., “Llama 2: Open Foundation and Fine-Tuned Chat Models,” arXiv preprint arXiv:2307.09288, 2023.

[13] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, “GQA: Training generalized multi-query transformer models from multi-head checkpoints,” in Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, H. Bouamor, J. Pino, and K. Bali, Eds., Singapore, Dec. 2023, pp. 4895–4901.

[14] A. Warstadt, A. Mueller, L. Choshen, E. Wilcox, C. Zhuang, J. Ciro, R. Mosquera, B. Paranjabe, A. Williams, T. Linzen, and R. Cotterell, “Findings of the BabyLM Challenge: Sample-Efficient Pretraining on Developmentally Plausible Corpora,” in Proc. BabyLM Challenge at the 27th Conf. Comput. Nat. Lang. Learn., Singapore, Dec. 2023, pp. 1–34.

[15] Ligeng Zhu, ”THOP: PyTorch-OpCounter”, Version 0.1.1, September, 2022. [Online]. Available: https://pypi.org/project/thop/

[16] A. Warstadt, A. Parrish, H. Liu, A. Mohananey, W. Peng, S.-F. Wang, and S. R. Bowman, “BLiMP: The Benchmark of Linguistic Minimal Pairs for English,” Transactions of the Association for Computational Linguistics, vol. 8, pp. 377–392, 2020.

[17] I. Tenney, D. Das, and E. Pavlick, “BERT Rediscovers the Classical NLP Pipeline,” in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, A. Korhonen, D. Traum, and L. Màrquez, Eds., Florence, Italy, Jul. 2019, pp. 4593–4601.

[18] I. Tenney, P. Xia, B. Chen, A. Wang, A. Poliak, R. T. McCoy, N. Kim, B. Van Durme, S. Bowman, D. Das, and E. Pavlick, “What do you learn from context? Probing for sentence structure in contextualized word representations,” in International Conference on Learning Representations, 2019.

[19] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov, “Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned,” arXiv preprint arXiv:1905.09418, 2019.