LLM Vocabulary Compression for Low-Compute Environments

Sreeram Vennam
IIIT Hyderabad

Anish Joishy
IIIT Hyderabad

Ponnurangam Kumaraguru
IIIT Hyderabad

Abstract

We present a method to compress the final linear layer of language models, reducing memory usage by up to 3.4x without significant performance loss. By grouping tokens based on Byte Pair Encoding (BPE) merges, we prevent materialisation of the memory-intensive logits tensor. Evaluations on the TinyStories dataset show that our method performs on par with GPT-Neo while significantly improving throughput by up to 3x, making it suitable for low-resource environments.

1 Introduction

The global trend in machine learning is increasingly focused on scaling with larger numbers of GPUs Musk (2024); Morgan (2024). In contrast, many researchers operate in low-compute environments, a disparity often referred to as the *compute divide* Besiroglu et al. (2024). Our work seeks to address this gap by optimising the utilisation of resource-constrained environments. Specifically, we target the vocabulary layer in language models. During training or fine-tuning, it becomes necessary to materialise a tensor of shape `[batch_size, sequence_length, vocab_size]`. Even with conservative values for the batch size and sequence length, such as a tensor of shape `[32, 512, 50000]`, this alone consumes approximately 3.32 GB of memory. For the remainder of this text, we will refer to this tensor as the *logits tensor*.

Although no research has directly addressed this specific tensor, several studies have acknowledged the vocabulary layer as a computational bottleneck and have proposed methods to reduce its computational complexity Jozefowicz et al. (2016). One of the earliest efforts was by Goodman (2001), who introduced a class-based approach by organising tokens into classes and training two separate models: one for predicting the class and another for predicting the actual token. Later, Joulin et al. (2017) proposed a hierarchical organisation of tokens based on their frequency, although computing these frequencies remains computationally expensive. Building on this, Baevski and Auli (2018) extended the hierarchical approach to allow for variable capacity inputs. However, a comprehensive analysis of memory usage in this context is still lacking.

In this work, we propose a method to reduce the memory footprint of the final embedding layer by grouping tokens and predicting the final token in a two-step process effectively compressing the vocabulary layer. Our approach differentiates from the work of Joulin et al. (2017); Goodman (2001) through two key innovations. First, rather than grouping tokens based on their frequency—a process that necessitates significant pre-processing—we group tokens according to the ordering of BPE Sennrich et al. (2016) merges, which inherently exploits token occurrence patterns. Second, we observe that it is unnecessary to use two separate models to predict the group and the token. Instead, simple linear layers over the hidden states can simultaneously learn both the group and token predictors. We empirically demonstrate that our modification does not negatively impact model performance.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

2 Methodology

2.1 Problem Formulation

Our goal is to produce a mapping from the hidden state h to a probability distribution over the entire vocabulary. The resulting probability distribution is used with `top_k`, sampling and other techniques during decoding.

$$P(v \mid h) = \text{Softmax}(Wh + b)$$

Where $P(v \mid h)$ is the probability distribution over the vocabulary given the hidden state h, W is the large weight matrix, and b is the bias vector.

Producing this distribution for every single token leads to materialising the logits tensor, therefore, to avoid this, during training, we want to materialise only a subset of this distribution, and during inference, we only need to produce this distribution for the last hidden state which is used to predict the next token in combination with techniques such as `top_k` and sampling.

2.2 Definitions and Notation

Grouping First, we divide the vocabulary into G groups based on their token indices. Each group contains tokens from consecutive index ranges. Let S_g and E_g be the starting index and ending index respectively for tokens part of group $g = 0, 1, \ldots, G-1$.

$$S_g = \left\lfloor \frac{|v| \cdot g}{G} \right\rfloor, \qquad E_g = \left\lfloor \frac{|v| \cdot (g+1)}{G} \right\rfloor - 1$$

For example, if we only had two groups; group 0 contains tokens from 0 to $\lfloor |v|/2 \rfloor - 1$ and group 1 contains tokens from $\lfloor |v|/2 \rfloor$ to $|v| - 1$. This partitioning implicitly makes use of BPE merging order, which is representative of the frequency of tokens.

Grouping Tensor We define weight block $W_g \in \mathbb{R}^{d \times G}$ where d is the hidden dimension of the transformer. This tensor is used to predict the group index from a hidden state.

Scale and Shift Tensors We define a 3 weight blocks, $W_s \in \mathbb{R}^{d \times S}$, $W_{Pg} \in \mathbb{R}^S$, and $W_{Qg} \in \mathbb{R}^S$ where S is the number of tokens per group and $g = 0, 1, \ldots, G-1$.

$W_s \in \mathbb{R}^{d \times S}$ it the shared linear tensor that is applied for all groups. $W_{Pg} \in \mathbb{R}^S$ is the scaling linear tensor specific to the group g. $W_{Qg} \in \mathbb{R}^S$ is the shifting linear tensor specific to the group g. These tensors are used to predict the exact token from the hidden state, once the group has been identified. We detail the operation in Section 2.3.1.

2.3 Method

Our method works differently during training vs inference since these modes of operation have different requirements. Training requires extensive parallelism but our knowledge of the labels tell us what group to choose. Inference requires the complete probability distribution over the entire vocabulary, but it requires it only for the last hidden state.

2.3.1 Training

During training, since we know the labels apriori, we already know what group each hidden state belongs to. Therefore, we only need to materialise the distribution for a token within a group, which is of size $S \leq v$.

We also train the grouping tensor during training, therefore, we have two objectives we are trying to minimise. The grouping tensor loss $\mathcal{L}_{\text{group}}$ coming from predicting which group each hidden state belongs to, and the token loss $\mathcal{L}_{\text{token}}$ coming from predicting tokens within a group.

Assume $h \in \mathbb{R}^d$ is a single hidden state of dimension d. We know that this hidden state belongs to group g and token t within group g. First, we apply the grouping tensor on h and calculate $\mathcal{L}_{\text{group}}$. To calculate $\mathcal{L}_{\text{token}}$, we first apply the linear block shared across groups W_s on h. We then apply the scale and shift tensors for group g which becomes $W_{Pg} \circ (W_s \cdot h) + W_{Qg}$. This tensor is then used against the label token t to produce $\mathcal{L}_{\text{token}}$. We refer to this scale and shift transformation as "applying linears". The final objective we minimise is the sum of both losses.

$$\mathcal{L}_{\text{group}} = \text{CrossEntropy}(W_g \cdot h, g),$$

$$\mathcal{L}_{\text{token}} = \text{CrossEntropy}(W_{Pg} \circ (W_s \cdot h) + W_{Qg}, t)$$

$$\mathcal{L} = \mathcal{L}_{\text{group}} + \mathcal{L}_{\text{token}},$$

Note that this transformation is not linear due to grouping, it forms a deeper network. An implementation for "applying linears" can be found in the Appendix A.

2.3.2 Inference

In the inference phase, the model generates the probability distribution over the entire vocabulary v from the last hidden state. We construct this distribution by using the token probability conditioning on the group probability.

Assume $h \in \mathbb{R}^d$ is a single hidden state of dimension d. We know that this hidden state belongs to group g and token t within group g. First, we apply the grouping tensor and `Softmax` to obtain a distribution over the groups for h. Then, for each group, we compute the token probability distribution by "applying linears" 2.3.1 and `Softmax` on h. We then multiply the token probabilities with their respective group probability. Finally we concatenate these distributions and the result is used in next token prediction.

$$\mathcal{P}_{\text{group}} = \text{Softmax}(W_g \cdot h, g),$$

$$\mathcal{P}_{\text{token}|g} = \text{Softmax}(W_{Pg} \cdot (W_s \cdot h) + W_{Qg}, t)$$

$$\mathcal{P}_{\text{vocab}} = \text{Concat}\left(\mathcal{P}_{\text{group}}[0] \cdot \mathcal{P}_{\text{token}|0}(t), \mathcal{P}_{\text{group}}[1] \cdot \mathcal{P}_{\text{token}|1}(t), \ldots, \mathcal{P}_{\text{group}}[G-1] \cdot \mathcal{P}_{\text{token}|G-1}(t)\right)$$

2.4 Optimal Memory Configuration

From the above description of our method, instead of materialising the logits tensor, we materialise `[batch_size, sequence_length, group_size]` and `[batch_size, sequence_length, num_groups]`. Let b be the batch size, s be the sequence length, S be the group size and G be the number of groups. We wish to minimise the memory usage of the combined tensors $[b, s, S]$ and $[b, s, G]$ under the condition that $G = \frac{v}{S}$. It is clear that this minima is achieved when $S = G = \sqrt{v}$, that is, we have $\sqrt{v}$ groups each containing $\sqrt{v}$ tokens.

3 Performance

3.1 Language Modelling

We test our approach on language modelling a task infamous for being infeasible on low compute devices. We apply our method on the GPT-2 architecture and compare memory usage against the base GPT-2 model and the GPT-Neo model.

Dataset We train on the TinyStories Eldan and Li (2023) dataset, the standard for testing small language models.

Metrics & Evaluation We adopt the LLM evaluation metrics from TinyStories. This includes evaluating grammar, creativity, consistency, plot coherency and the estimated age of the writer.

Setting We replicate the settings used in TinyStories, we use batch size 32 with sequence length 512. GPT-Neo uses a window size of 256. We don't however use a trained tokenizer since the vocabulary size was relatively small and isn't much of a bottleneck.

We train 3 different sized models based on hidden size per architecture to extensively evaluate our approach. Every model uses 8 layers and 8 heads per layer. All models were trained for a single epoch over the entire dataset. This is because we had limited access to compute and training multiple models for several epochs would take months for the compute we had available. Loss curves can be found in the Appendix B.1.

Table 1: Comparing Model Performance on TinyStories

Model	Hidden Size	Grammar	Creativity	Consistency	Plot	Age
GPT-2	128	3.8	4.22	3.7	2.95	4.55
	256	4.65	3.6	6.05	4.82	4.75
	512	5.82	3.9	7.15	5.75	4.9
GPT-Neo	128	3.7	3.72	4.05	2.97	4.5
	256	4.77	4.1	5.85	4.57	4.65
	512	5.15	4.17	6.65	5.22	5.0
Ours	128	3.53	3.62	4.5	3.25	4.55
	256	4.85	4.3	5.87	4.47	4.7
	512	5.15	4.025	6.7	5.475	4.96

Results Table 1 contains the results for the models we trained. First we see that model performance does increase with model size validating our setup. Our approach performs on par with GPT-Neo and GPT-2. These results strongly suggest that our approach is able to compress the vocabulary layer without significant loss in performance.

3.2 Multiclass Classification

For robustness, we evaluate our approach on a task disconnected from language modelling, image classification. Here grouping is arbitrary, but even in such conditions, our approach works surprisingly well. The implementation details for this experiment including the architecture used can be found in the Appendix C.

Dataset We generate 100×100 images with different attributes and style, each such combination of attributes is a label. We produce a synthetic dataset since real world datasets don't often have a large number of labels. Our dataset has 184320 unique labels. Some examples from the dataset are presented in Figure 1. A comprehensive description of the dataset is present in the Appendix C.1.

Figure 1: Example images from the synthetic dataset for image classification.

Results Figure 2 plots validation accuracy across training steps. We find that our approach enables learning in scenarios in which learning would otherwise not have occurred despite using significantly fewer parameters. Our method improves performance since the group predictor learns C.4

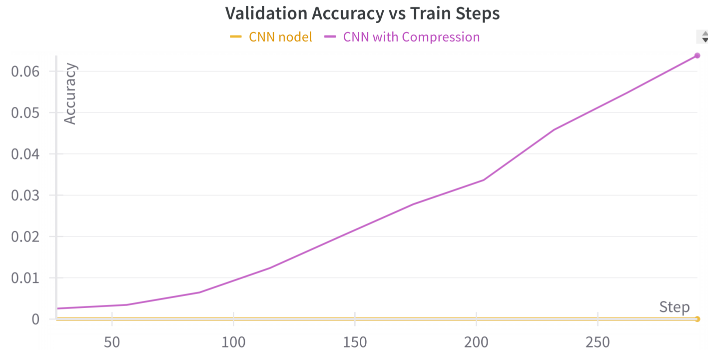

Figure 2: Validation accuracy vs Train steps for the synthetic image classification dataset.

4 Memory Usage

We empirically verify that our approach significantly saves on memory and is upto 3.4x times more efficient in certain scenarios. Table 2 contains our report for memory used by different models.

Setting We monitor the memory usage during a short training epoch (100 batches) and a single validation epoch. We report the peak memory reserved by the program for training. We analyze 4 models per architecture with varying hidden sizes. All models use 8 layers and 8 heads per layer. The model of size 8.1 M uses $h = 128$, the 19.3 M uses $h = 256$, the 152 M uses $h = 1024$, and the 659 M uses $h = 2048$.

Results Table 2 contains the results of this experiment, "oom" stands for out of memory. We see that we achieve non-trivial memory efficiency simply by compressing the vocabulary layer. Our results show that we can double (T4) or in some cases triple (A10G) the model size and still train the model on the same GPU.

Table 2: Memory Usage Comparison for GPT-2, GPT-Neo, and Our Implementation on TinyStories during a training loop. "oom" stands for out of memory.

		Memory Usage (GB)			
GPU	Model Size	GPT-2	GPT-Neo	Ours	Efficiency
T4	8.1 M	oom	13.0	**3.80**	3.4x
	19.3 M	oom	oom	**5.20**	na
	152 M	oom	oom	oom	na
	659 M	oom	oom	oom	na
A10G	8.1 M	14.0	13.0	**3.80**	3.4x
	19.3 M	16.0	18.0	**5.20**	3.4x
	152 M	oom	oom	**16.6**	na
	659 M	oom	oom	oom	na
L40s	8.1 M	14.0	13.0	**3.80**	3.4x
	19.3 M	16.0	18.0	**5.20**	3.4x
	152 M	28.0	30.0	**16.6**	1.8x
	659 M	oom	oom	**39.0**	na

5 Computational Efficiency

Our model significantly reduces the number of parameters required in the vocabulary layer. This not only reduces the average flops, but it also significantly speeds up the performance of the forward pass. The settings for this experiment can be found in the Appendix E.

Results Table 3 showcases our results for performance in terms of throughput and flops. We see a staggering 3x (8.1 M) increase in throughput and a 5x (8.1 M) decrease in FLOPs used all while maintaining performance on language modelling. We see that the improvement decreases with increase in model size as the vocabulary layer becomes less of a bottleneck.

Table 3: Token Throughput and FLOPs Comparison for GPT-2, GPT-Neo, and Our Implementation on a single L40s.

	Throughput (Tokens/s)			**FLOPs** (GFLOPs)		
Model Size	GPT-2	GPT-Neo	Ours	GPT-2	GPT-Neo	Ours
8.1 M	108373	80757	313049	4.10	4.64	0.83
19.3 M	90458	69967	234268	9.00	10.0	3.30
152 M	32194	31523	59106	72.0	82.0	52.0

6 Ablation Studies

The group size is a key hyper parameter that can greatly impact the performance and efficiency of our method. We perform an ablation study varying the group size while monitoring $\mathcal{L}_{\text{val}}$ and memory used. Implementation details for this study can be found in the Appendix D.

Results Figure 3 showcases the study. As discussed in Section 2.4, we see optimal memory usage when group size is equal to the square root of the vocabulary. We also see that this is the maxima in terms of val loss. This makes sense as we have the most inter-group confusion at this point, however, the performance difference isn't very large varying between 3.7 and 3.9 for most group sizes around $\sqrt{v}$.

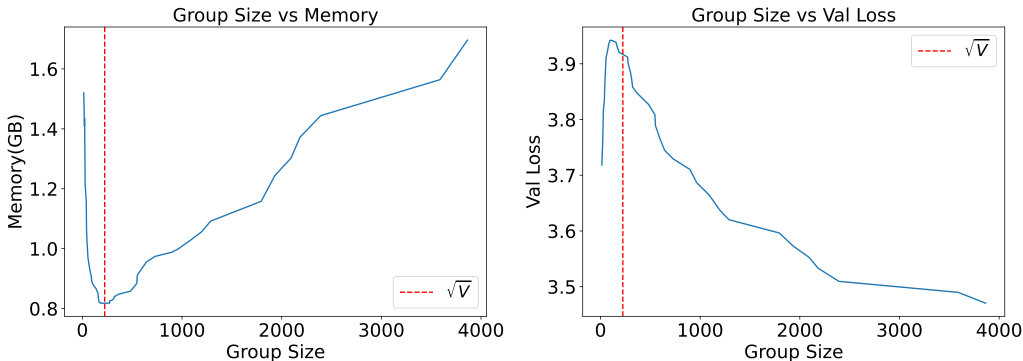

Figure 3: Ablation studies

7 Limitations & Conclusion

While our method improves memory efficiency and computational performance, several limitations remain. Despite surpassing GPT-2 and GPT-Neo models, we were unable to compare with other contemporary vocabulary compression techniques due to resource constraints. Additionally, our method shows sensitivity to the group size hyperparameter, as noted in our ablation study. Limited training epochs also impacts results, but we believe they are representative of low-compute settings.

In this work, we introduced a novel approach to compressing the final vocabulary layer, achieving both memory efficiency and high performance. We hope this contributes to low-compute machine learning and demonstrates how simple optimisations can enhance low-compute environments.

References

Baevski, A. and Auli, M. (2018). Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*.

Besiroglu, T., Bergerson, S. A., Michael, A., Heim, L., Luo, X., and Thompson, N. (2024). The compute divide in machine learning: A threat to academic contribution and scrutiny?

Eldan, R. and Li, Y. (2023). Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*.

Goodman, J. (2001). Classes for fast maximum entropy training. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, volume 1, pages 561–564. IEEE.

Joulin, A., Cissé, M., Grangier, D., Jégou, H., et al. (2017). Efficient softmax approximation for gpus. In *International conference on machine learning*, pages 1302–1310. PMLR.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling.

Morgan, T. P. (2024). Inside the massive gpu buildout at meta platforms. *The Next Platform*. Accessed: 2024-10-01.

Musk, E. (2024). Elon musk buys thousands of gpus for twitter's generative ai project. *Tom's Hardware*.

Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

A Psuedocode

An alternative that we initially explored involved storing a unique linear tensor for each group (Figure 4) instead of a shared linear with a scale and shift (Figure 5. This forced us to loop over the groups if we wished to keep memory usage low. Even with a clever implementation using masks, this approach was significantly slower and was replaced with the scale and shift tensors.

```
def apply_linear(self, h, groups):
    batch_size, sequence_length, hidden_size = h.shape
    output = torch.zeros(batch_size, sequence_length,
        self.group_size, device=h.device)

    h_flat = h.view(-1, hidden_size)
    output_flat = output.view(-1, self.group_size)
    groups_flat = groups.view(-1)

    for i in range(self.num_groups):
        mask = (groups_flat == i)
        if mask.any():
            group_input = h_flat[mask]
            group_output = self.linears[i](group_input)
            output_flat[mask] = group_output

    return output_flat.view(batch_size, sequence_length, self.group_size)
```

Figure 4: Slow implementation which requires looping over the groups

```
def apply_linear(self, h, groups):
    shared_output = self.shared_linear(h)

    groups_flat = groups.view(-1)
    shared_output_flat = shared_output.view(-1, self.group_size)

    scale = self.scale(groups_flat)
    shift = self.shift(groups_flat)

    modulated_output_flat = shared_output_flat * scale + shift
    modulated_output = modulated_output_flat.view_as(shared_output)

    return modulated_output
```

Figure 5: Fast implementation applying the scale and shift transformation.

B TinyStories

B.1 Loss Curves

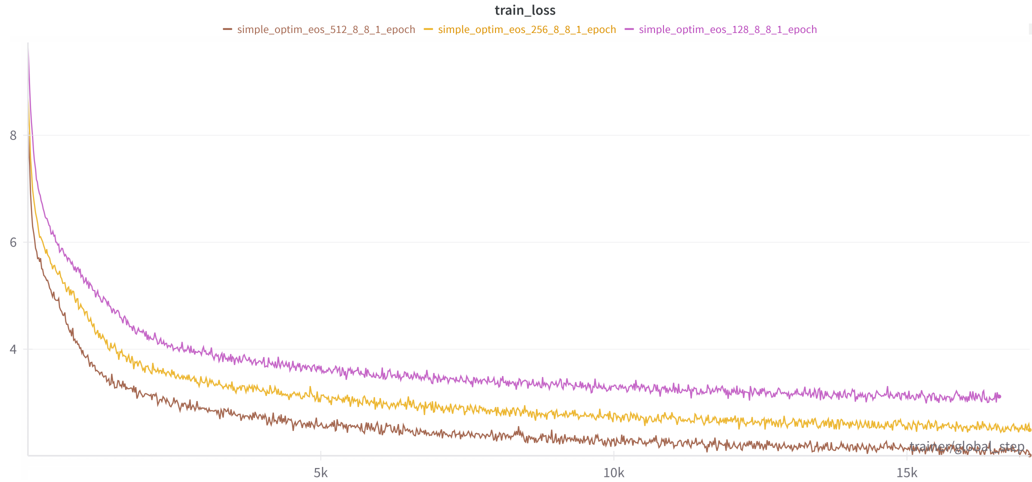

Figure 6: Train loss curves for our models on TinyStories

C Multi Class Classification

C.1 Dataset

The Dataset comprised of 184k classes each containing about 10 images. Each class comprised of the following attributes.

Shapes: {Circle, Square, Triangle, Pentagon, Hexagon, Octagon, Star, Cross}

Patterns: {Solid, Striped, Dotted, Checkered}

Rotations: {0°, 60°, 120°, 180°, 240°, 300°}

Colors: {Red, Green, Blue, Yellow, Purple, Orange, Pink, Brown, Cyan, Magenta, Lime, Teal}

Sizes: {Tiny, Small, Medium, Large, Huge}

Textures: {None, Noise}

Opacities: {0.25, 0.5, 0.75, 1.0}

Border Styles: {Solid, Dashed}

One attribute from each class was randomly chosen to generate about 2M images. Every possible class was mapped to an index and stored along with the images.

C.2 Model Architecture

We used a 3 layer CNN with Kernel Size = 3 and padding = 1. Channels were taken from 3 to 32 in the first layer and doubled in each subsequent layer. The output was flattened and subsequently encoded into a 512-dimensional embedding through a fully connected layer along with Relu non linearity.

The baseline model linearly mapped this to the vocab size, whereas our method mapped it into one of the $\sqrt{V}$ groups and subsequently mapped to a particular token.

Parameters Count of the baseline model: **104,087,552**

Parameters Count of the our model: **10,340,999**

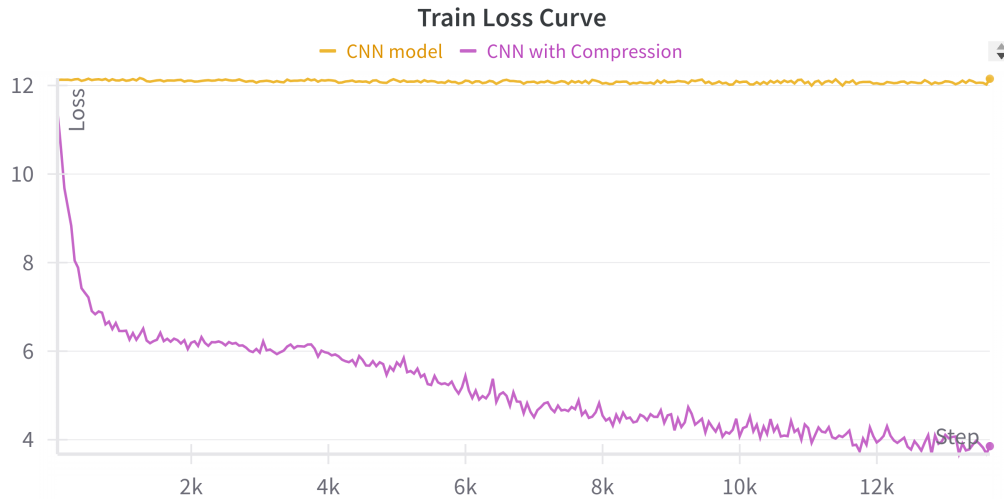

Figure 7: Train loss curve comparison on multi-class classification

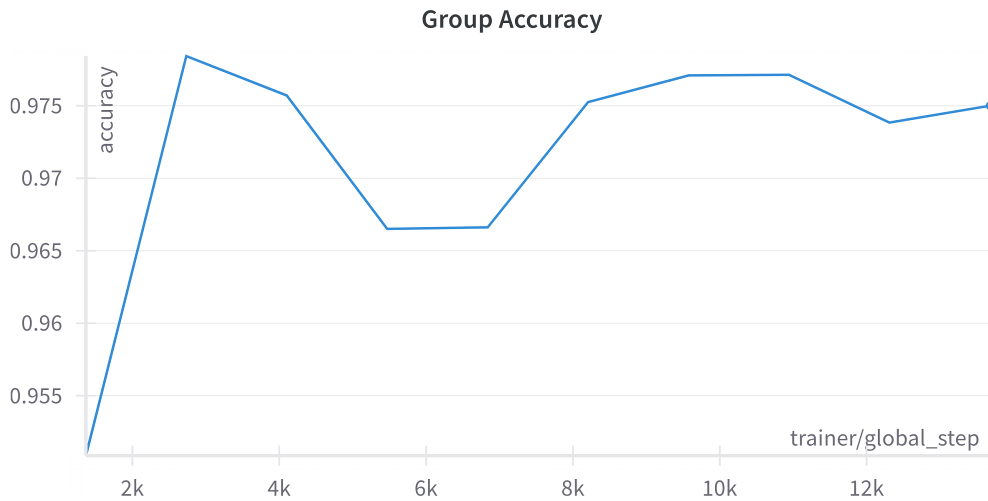

Figure 8: Group prediction accuracy of our model. We see that out model takes advantage of the fact that smaller groups are easier to learn than the whole dataset

C.3 Training

The training was conducted for 10 epochs with a batch size of 64 and gradient accumulation over 4 batches. Learning rate was set to 0.001 and Adam optimizer was used to update the weights. Cross Entropy Loss was used to train and evaluate the model's performance. Accuracy was recorded at the end of each epoch over the Validation set.

C.4 Train Plots

D Ablation Studies

To study the effect of group sizes on memory and model performance, small models with 64 dimensional embeddings with 4 layers and 2 heads were trained on the TinyStories dataset for half an epoch each. Vocabulary Size, V was equal to 50258.

To ensure equal distribution of group sizes above and below $\sqrt{V}$, values of group sizes were taken as the factors of 50232 as this number has a lot of factors and is close to 50258. 8 factors at the extremities were not considered as these would lead to very few groups or very few elements in a group. Around 50 models were trained under these settings, and memory utilized and Validation losses were recorded for each model.

E Computational Efficiency

Similar to memory usage, we run a single short training loop and report the average throughput in terms of tokens per second. We also report the average FLOPs used per forward pass which is another strong measure of computational efficiency. For measuring FLOPs, we use a single sample of sequence length 512.

This was done on TinyStories for a batch size of 32 and sequence length of 512.