Scaling Deep Contrastive Learning Batch Size under Memory Limited Setup

Luyu Gao[1], Yunyi Zhang[2], Jiawei Han[2], Jamie Callan[1]
[1] Language Technologies Institute, Carnegie Mellon University
[2] Department of Computer Science, University of Illinois Urbana-Champaign
[1]{luyug, callan}@cs.cmu.edu   [2]{yzhan238, hanj}@illionis.edu

Abstract

Contrastive learning has been applied successfully to learn vector representations of text. Previous research demonstrated that learning high-quality representations benefits from batch-wise contrastive loss with a large number of negatives. In practice, the technique of in-batch negative is used, where for each example in a batch, other batch examples' positives will be taken as its negatives, avoiding encoding extra negatives. This, however, still conditions each example's loss on all batch examples and requires fitting the entire large batch into GPU memory. This paper introduces a gradient caching technique that decouples backpropagation between contrastive loss and the encoder, removing encoder backward pass data dependency along the batch dimension. As a result, gradients can be computed for one subset of the batch at a time, leading to almost constant memory usage. [1]

1 Introduction

Contrastive learning learns to encode data into an embedding space such that related data points have closer representations and unrelated ones have further apart ones. Recent works in NLP adopt deep neural nets as encoders and use unsupervised contrastive learning on sentence representation (Giorgi et al., 2020), text retrieval (Lee et al., 2019), and language model pre-training tasks (Wu et al., 2020). Supervised contrastive learning (Khosla et al., 2020) has also been shown effective in training dense retrievers (Karpukhin et al., 2020; Qu et al., 2020). These works typically use batch-wise contrastive loss, sharing target texts as in-batch negatives. With such a technique, previous works have empirically shown that larger batches help learn better representations. However, computing loss and updating model parameters with respect

to a big batch require encoding all batch data and storing all activation, so batch size is limited by total available GPU memory. This limits application and research of contrastive learning methods under memory limited setup, e.g. academia. For example, Lee et al. (2019) pre-train a BERT (Devlin et al., 2019) passage encoder with a batch size of 4096 while a high-end commercial GPU RTX 2080ti can only fit a batch of 8. The gradient accumulation technique, splitting a large batch into chunks and summing gradients across several backwards, cannot emulate a large batch as each smaller chunk has fewer in-batch negatives.

In this paper, we present a simple technique that thresholds peak memory usage for contrastive learning to almost constant regardless of the batch size. For deep contrastive learning, the memory bottlenecks are at the deep neural network based encoder. We observe that we can separate the backpropagation process of contrastive loss into two parts, from loss to representation, and from representation to model parameter, with the latter being independent across batch examples given the former, detailed in subsection 3.2. We then show in subsection 3.3 that by separately pre-computing the representations' gradient and store them in a cache, we can break the update of the encoder into multiple sub-updates that can fit into the GPU memory. This pre-computation of gradients allows our method to produce the *exact same* gradient update as training with large batch. Experiments show that with about 20% increase in runtime, our technique enables a single consumer-grade GPU to reproduce the state-of-the-art large batch trained models that used to require multiple professional GPUs.

2 Related Work

Contrastive Learning First introduced for probablistic language modeling (Mnih and Teh, 2012),

---

[1] Our code is at github.com/luyug/GradCache.

Noise Contrastive Estimation (NCE) was later used by Word2Vec (Mikolov et al., 2013) to learn word embedding. Recent works use contrastive learning to unsupervisedly pre-train (Lee et al., 2019; Chang et al., 2020) as well as supervisedly train dense retriever (Karpukhin et al., 2020), where contrastive loss is used to estimate retrieval probability over the entire corpus. Inspired by SimCLR (Chen et al., 2020), constrastive learning is used to learn better sentence representation (Giorgi et al., 2020) and pre-trained language model (Wu et al., 2020).

Deep Network Memory Reduction Many existing techniques deal with large and deep models. The gradient checkpoint method attempts to emulate training deep networks by training shallower layers and connecting them with gradient checkpoints and re-computation (Chen et al., 2016). Some methods also use reversible activation functions, allowing internal activation in the network to be recovered throughout back propagation (Gomez et al., 2017; MacKay et al., 2018). However, their effectiveness as part of contrastive encoders has not been confirmed. Recent work also attempts to remove the redundancy in optimizer tracked parameters on each GPU (Rajbhandari et al., 2020). Compared with the aforementioned methods, our method is designed for scaling over the batch size dimension for contrastive learning.

3 Methodologies

In this section, we formally introduce the notations for contrastive loss and analyze the difficulties of using it on limited hardware. We then show how we can use a Gradient Cache technique to factor the loss so that large batch gradient update can be broken into several sub-updates.

3.1 Preliminaries

Under a general formulation, given two classes of data $\mathcal{S}, \mathcal{T}$, we want to learn encoders f and g for each such that, given $s \in \mathcal{S}, t \in \mathcal{T}$, encoded representations $f(s)$ and $g(t)$ are close if related and far apart if not related by some distance measurement. For large $\mathcal{S}$ and $\mathcal{T}$ and deep neural network based f and g, direct training is not tractable, so a common approach is to use a contrastive loss: sample anchors $S \subset \mathcal{S}$ and targets $T \subset \mathcal{T}$ as a training batch, where each element $s_i \in S$ has a related element $t_{r_i} \in T$ as well as zero or more specially sampled hard negatives. The rest of the random samples in T will be used as in-batch negatives.

Define loss based on dot product as follows:

$$\mathcal{L} = -\frac{1}{|S|} \sum_{s_i \in S} \log \frac{exp(f(s_i)^\top g(t_{r_i})/\tau)}{\sum_{t_j \in T} exp(f(s_i)^\intercal g(t_j)/\tau)} \quad (1)$$

where each summation term depends on the *entire* set T and requires fitting *all* of them into memory.

We set temperature $\tau = 1$ in the following discussion for simplicity as in general it only adds a constant multiplier to the gradient.

3.2 Analysis of Computation

In this section, we give a mathematical analysis of contrastive loss computation and its gradient. We show that the back propagation process can be divided into two parts, from loss to representation, and from representation to encoder model. The separation then enables us to devise a technique that removes data dependency in encoder parameter update. Suppose the function f is parameterized with Θ and g is parameterized with Λ.

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \sum_{s_i \in S} \frac{\partial \mathcal{L}}{\partial f(s_i)} \frac{\partial f(s_i)}{\partial \Theta} \quad (2)$$

$$\frac{\partial \mathcal{L}}{\partial \Lambda} = \sum_{t_j \in T} \frac{\partial \mathcal{L}}{\partial g(t_j)} \frac{\partial g(t_j)}{\partial \Lambda} \quad (3)$$

As an extra notation, denote normalized similarity,

$$p_{ij} = \frac{exp(f(s_i)^\intercal g(t_j))}{\sum_{t \in T} exp(f(s_i)^\intercal g(t))} \quad (4)$$

We note that the summation term for a particular s_i or t_i is a function of the batch, as,

$$\frac{\partial \mathcal{L}}{\partial f(s_i)} = -\frac{1}{|S|} \left(g(t_{r_i}) - \sum_{t_j \in T} p_{ij} g(t_j) \right), \quad (5)$$

$$\frac{\partial \mathcal{L}}{\partial g(t_j)} = -\frac{1}{|S|} \left(\epsilon_j - \sum_{s_i \in S} p_{ij} f(s_i) \right), \quad (6)$$

where

$$\epsilon_j = \begin{cases} f(s_k) & \text{if } \exists\, k \text{ s.t. } r_k = j \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

which prohibits the use of gradient accumulation. We make two observations here:

- The partial derivative $\frac{\partial f(s_i)}{\partial \Theta}$ depends only on s_i and Θ while $\frac{\partial g(t_j)}{\partial \Lambda}$ depends only on t_j and Λ; and

317

- Computing partial derivatives $\frac{\partial \mathcal{L}}{\partial f(s_i)}$ and $\frac{\partial \mathcal{L}}{\partial g(t_j)}$ requires only encoded representations, but not Θ or Λ.

These observations mean back propagation of $f(s_i)$ for data s_i can be run independently with its own computation graph and activation if the *numerical* value of the partial derivative $\frac{\partial \mathcal{L}}{\partial s_i}$ is known. Meanwhile the derivation of $\frac{\partial \mathcal{L}}{\partial s_i}$ requires only *numerical* values of two sets of representation vectors $F = \{f(s_1), f(s_2), .., f(s_{|S|})\}$ and $G = \{g(t_1), g(t_2), ..., g(t_{|T|})\}$. A similar argument holds true for g, where we can use representation vectors to compute $\frac{\partial \mathcal{L}}{\partial t_j}$ and back propagate for each $g(t_j)$ independently. In the next section, we will describe how to scale up batch size by pre-computing these representation vectors.

3.3 Gradient Cache Technique

Given a large batch that does not fit into the available GPU memory for training, we first divide it into a set of sub-batches each of which can fit into memory for gradient computation, denoted as $\mathbb{S} = \{\hat{S}_1, \hat{S}_2, ..\}, \mathbb{T} = \{\hat{T}_1, \hat{T}_2, ..\}$. The full-batch gradient update is computed by the following steps.

Step1: Graph-less Forward Before gradient computation, we first run an extra encoder forward pass for each batch instance to get its representation. Importantly, this forward pass runs without constructing the computation graph. We collect and store all representations computed.

Step2: Representation Gradient Computation and Caching We then compute the contrastive loss for the batch based on the representation from Step1 and have a corresponding computation graph constructed. Despite the mathematical derivation, automatic differentiation system is used in actual implementation, which automatically supports variations of contrastive loss. A backward pass is then run to populate gradients for each representation. Note that the encoder is not included in this gradient computation. Let $\mathbf{u}_i = \frac{\partial \mathcal{L}}{\partial f(s_i)}$ and $\mathbf{v}_i = \frac{\partial \mathcal{L}}{\partial g(t_i)}$, we take these gradient tensors and store them as a *Representation Gradient Cache*, $[\mathbf{u}_1, \mathbf{u}_2, .., \mathbf{v}_1, \mathbf{v}_2, ..]$.

Step3: Sub-batch Gradient Accumulation We run encoder forward one sub-batch at a time to compute representations and build the corresponding computation graph. We take the sub-batch's representation gradients from the cache and run

back propagation through the encoder. Gradients are accumulated for encoder parameters across all sub-batches. Effectively for f we have,

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \Theta} &= \sum_{\hat{S}_j \in \mathbb{S}} \sum_{s_i \in \hat{S}_j} \frac{\partial \mathcal{L}}{\partial f(s_i)} \frac{\partial f(s_i)}{\partial \Theta} \\
&= \sum_{\hat{S}_j \in \mathbb{S}} \sum_{s_i \in \hat{S}_j} \mathbf{u}_i \frac{\partial f(s_i)}{\partial \Theta}
\end{aligned}
\tag{8}
$$

where the outer summation enumerates each sub-batch and the entire internal summation corresponds to one step of accumulation. Similarly, for g, gradients accumulate based on,

$$
\frac{\partial \mathcal{L}}{\partial \Lambda} = \sum_{\hat{T}_j \in \mathbb{T}} \sum_{t_i \in \hat{T}_j} \mathbf{v}_i \frac{\partial g(t_i)}{\partial \Lambda}
\tag{9}
$$

Here we can see the *equivalence* with direct large batch update by combining the two summations.

Step4: Optimization When all sub-batches are processed, we can step the optimizer to update model parameters as if the full batch is processed in a single forward-backward pass.

Compared to directly updating with the full batch, which requires memory linear to the number of examples, our method fixes the number of examples in each encoder gradient computation to be the size of sub-batch and therefore requires constant memory for encoder forward-backward pass. The extra data pieces introduced by our method that remain persistent across steps are the representations and their corresponding gradients with the former turned into the latter after representation gradient computation. Consequently, in a general case with data from S and T each represented with d dimension vectors, we only need to store $(|S|d + |T|d)$ floating points in the cache on top of the computation graph. To remind our readers, this is several orders smaller than million-size model parameters.

3.4 Multi-GPU Training

When training on multiple GPUs, we need to compute the gradients with all examples across all GPUs. This requires a single additional cross GPU communication after *Step1* when all representations are computed. We use an all-gather operation to make all representations available on all GPUs. Denote F^n, G^n representations on n-th GPU and a total of N device. *Step2* runs with gathered representations $F^{\text{all}} = F^1 \cup .. \cup F^N$ and $G^{\text{all}} = G^1 \cup .. \cup G^N$. While F^{all} and G^{all} are used

Method	Top-5	Top-20	Top-100
DPR	-	78.4	85.4
Sequential	59.3	71.9	80.9
Accumulation	64.3	77.2	84.9
Cache	68.6	79.3	86.0
- BSZ = 512	68.3	79.9	86.6

Table 1: Retrieval: We compare top-5/20/100 hit accuracy of small batch update (Sequential), accumulated small batch (Accumulation) and gradient cache (Cache) systems with DPR reference.

to compute loss, the n-th GPU only computes gradient of its local representations F^n, G^n and stores them into cache. No communication happens in *Step3*, when each GPU independently computes gradient for local representations. *Step4* will then perform gradient reduction across GPUs as with standard parallel training.

4 Experiments

To examine the reliability and computation cost of our method, we implement our method into dense passage retriever (DPR; Karpukhin et al. (2020))[2]. We use gradient cache to compute DPR's supervised contrastive loss on a single GPU. Following DPR paper, we measure top hit accuracy on the Natural Question Dataset (Kwiatkowski et al., 2019) for different methods. We then examine the training speed of various batch sizes.

4.1 Retrieval Accuracy

Compared Systems 1) **DPR**: the reference number taken from the original paper trained on 8 GPUs, 2) **Sequential**: update with max batch size that fits into 1 GPU, 3) **Accumulation**: similar to Sequential but accumulate gradients and update until number of examples matches DPR setup, 4) **Cache**: training with DPR setup using our gradient cache on 1 GPU. We attempted to run with gradient checkpointing but found it cannot scale to standard DPR batch size on our hardware.

Implementations All runs start with the same random seed and follow DPR training hyperparameters except batch size. Cache uses a batch size of 128 same as DPR and runs with a sub-batch size of 16 for questions and 8 for passages. We also run Cache with a batch size of 512 (BSZ=512) to
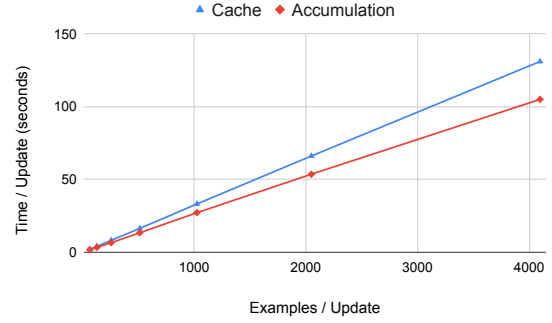
Figure 1: We compare training speed versus the number of examples per update for gradient cache (Cache) and gradient accumulation (Accumulation).

examine the behavior of even larger batches. Sequential uses a batch size of 8, the largest that fits into memory. Accumulation will accumulate 16 of size-8 batches. Each question is paired with a positive and a BM25 negative passage. All experiments use a single RTX 2080ti.

Results Accuracy results are shown in Table 1. We observe that Cache performs better than DPR reference due to randomness in training. Further increasing batch size to 512 can bring in some advantage at top 20/100. Accumulation and Sequential results confirm the importance of a bigger batch and more negatives. For Accumulation which tries to match the batch size but has fewer negatives, we see a drop in performance which is larger towards the top. In the sequential case, a smaller batch incurs higher variance, and the performance further drops. In summary, our Cache method improves over standard methods and matches the performance of large batch training.

4.2 Training Speed

In Figure 1, we compare update speed of gradient cache and accumulation with per update example number of $\{64, 128, 256, 512, 1024, 2048, 4096\}$. We observe gradient cache method can steadily scale up to larger batch update and uses 20% more time for representation pre-computation. This extra cost enables it to create an update of a much larger batch critical for the best performance, as shown by previous experiments and many early works. While the original DPR reports a training time of roughly one day on 8 V100 GPUs, in practice, with improved data loading, our gradient cache code can train a dense retriever in a practical 31 hours on a single RTX2080ti. We also find gradient checkpoint only runs up to batch of 64 and consumes

twice the amount of time than accumulation[3].

5 Extend to Deep Distance Function

Previous discussion assumes a simple parameterless dot product similarity. In general it can also be deep distance function Φ richly parameterized by Ω, formally,

$$d_{ij} = d(s_i, t_j) = \Phi(f(s_i), g(t_j)) \qquad (10)$$

This can still scale by introducing an extra *Distance Gradient Cache*. In the first forward we collect all representations as well as all distances. We compute loss with d_{ij}s and back propagate to get $w_{ij} = \frac{\partial \mathcal{L}}{\partial d_{ij}}$, and store them in Distance Gradient Cache, $[w_{00}, w_{01}, .., w_{10}, ..]$. We can then update Ω in a sub-batch manner,

$$\frac{\partial \mathcal{L}}{\partial \Omega} = \sum_{\hat{S} \in \mathbb{S}} \sum_{\hat{T} \in \mathbb{T}} \sum_{s_i \in \hat{S}} \sum_{t_j \in \hat{T}} w_{ij} \frac{\partial \Phi(f(s_i), g(t_j))}{\partial \Omega}$$

$$(11)$$

Additionally, we *simultaneously* compute with the constructed computation graph $\frac{\partial d_{ij}}{\partial f(s_i)}$ and $\frac{\partial d_{ij}}{\partial g(t_j)}$ and accumulate across batches,

$$\mathbf{u}_i = \frac{\partial \mathcal{L}}{\partial f(s_i)} = \sum_j w_{ij} \frac{\partial d_{ij}}{\partial f(s_i)} \qquad (12)$$

and,

$$\mathbf{v}_j = \frac{\partial \mathcal{L}}{\partial g(t_j)} = \sum_i w_{ij} \frac{\partial d_{ij}}{\partial g(t_j)} \qquad (13)$$

with which we can build up the Representation Gradient Cache. When all representations' gradients are computed and stored, encoder gradient can be computed with *Step3* described in subsection 3.3. In philosophy this method links up two caches. Note this covers early interaction $f(s) = s, g(t) = t$ as a special case.

6 Conclusion

In this paper, we introduce a gradient cache technique that breaks GPU memory limitations for large batch contrastive learning. We propose to construct a representation gradient cache that removes in-batch data dependency in encoder optimization. Our method produces the exact same gradient update as training with a large batch. We show the

method is efficient and capable of preserving accuracy on resource-limited hardware. We believe a critical contribution of our work is providing a large population in the NLP community with access to batch-wise contrastive learning. While many previous works come from people with industry-grade hardware, researchers with limited hardware can now use our technique to reproduce state-of-the-art models and further advance the research without being constrained by available GPU memory.

Acknowledgments

---

[3]We used the gradient checkpoint implemented in Huggingface transformers package

References

Wei-Cheng Chang, Felix X. Yu, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar. 2020. Pre-training tasks for embedding-based large-scale retrieval. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

T. Chen, B. Xu, C. Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *ArXiv*, abs/1604.06174.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. A simple framework for contrastive learning of visual representations. *ArXiv*, abs/2002.05709.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

John Michael Giorgi, Osvald Nitski, Gary D Bader, and Bo Wang. 2020. Declutr: Deep contrastive learning for unsupervised textual representations. *ArXiv*, abs/2006.03659.

Aidan N. Gomez, Mengye Ren, R. Urtasun, and Roger B. Grosse. 2017. The reversible residual network: Backpropagation without storing activations. In *NIPS*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *arXiv preprint arXiv:2004.11362*.

T. Kwiatkowski, J. Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, C. Alberti, D. Epstein, Illia Polosukhin, J. Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Q. Le, and Slav Petrov. 2019. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6086–6096, Florence, Italy. Association for Computational Linguistics.

Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger B. Grosse. 2018. Reversible recurrent neural networks. In *NeurIPS*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, G. Corrado, and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*.

A. Mnih and Y. Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. In *ICML*.

Yingqi Qu, Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. 2020. Rocketqa: An optimized training approach to dense passage retrieval for open-domain question answering.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models.

Z. Wu, Sinong Wang, Jiatao Gu, Madian Khabsa, Fei Sun, and Hao Ma. 2020. Clear: Contrastive learning for sentence representation. *ArXiv*, abs/2012.15466.