FastTran: Improving Performance of Fine-tuning in Transformer Models

Akshata Bhat
akshatabhat@cs.wisc.edu

Rohit Kumar Sharma
rsharma@cs.wisc.edu

Kyle Klassy
klassy@cs.wisc.edu

October 4, 2020

Abstract

In recent years, Transformer models have been widely used for various Natural Language Processing tasks and have achieved state-of-the-art accuracy. However, these models are computationally expensive and require a lot of time even for fine-tuning. In this work, we use various profiling techniques to understand the workload patterns of Transformer models. We introduce FastTran, a system that uses a caching mechanism to reduce the fine-tuning time for Transformer models. We also propose some future research directions that could further increase performance and reduce times for this workload.

1 Introduction

Recurrent Neural Networks such as LSTMs [9] and GRUs [1] have been widely used by the Natural Language Processing community to address many downstream tasks such as sentiment classification, question answering, etc., because of their nice property of having the ability to remember a hidden state for long sentences and paragraphs. Building good language models is crucial for achieving good performance in these tasks. But RNNs, due to their architecture, have two problems: the model forgets the hidden state of distant positions, and they are hard to train in parallel. To address these challenges, there have been breakthroughs and fast-paced developments in the field over the last few years. Namely, NLP scientists have developed a new architecture called Transformers [29]. Intuitively, the Transformer architecture invents a mechanism to pay attention to specific words. For example, during Neural Machine Translation, attention is paid to the word and its context in the source language that is currently being translated. Using a self-attention mechanism, the models can then remember long-term dependencies. By not relying on recurrent neural network modules, Transformers can be efficiently parallelized. Taking this idea one step further, the Transformers are trained in a bi-directional architecture for better language modeling by capturing context from both directions in BERT [3]. The pre-trained BERT model is widely used in downstream tasks by fine-tuning to the target task and dataset. More recently (XLM [15], GPT-2 [24], XLNet [33], RoBERT [17]), this architecture has been improved using various other Transformer models to further address some nuances.

Though these models have succeeded in creating new benchmarks in many NLP tasks, they are very inefficient to train from scratch. Because of the sheer number of parameters in these models (BERT-large has 340 million parameters) and the huge amount of data they need, training them takes a long time (multiple days) and requires many resources (CPUs, GPUs, memory, etc.). Therefore, the general pipeline of how these models are used is as follows: In the first phase, the model is trained only once using a lot of resources and training data such as Wikipedia and BookCorpus in a self-supervised manner (*Pre-training phase*). In the second phase, for each downstream task, the pre-trained model parameters are directly used and the model is fine-tuned end-to-end. In this phase, the parameters are allowed to adjust to "fit" to the training data (*Fine-tuning phase*). In the final inference phase, the fine-tuned model is used to make predictions [1] (*Inference phase*). Because these complex models are pre-trained only once, the researchers don't have to worry about the performance in the pre-training phase. In fact, pre-trained parameters can be directly downloaded for most models, which the researchers can then use as a starting point for the fine-tuning phase. However, since these models are large, even fine-tuning the parameters end-to-end for a specific downstream task is not trivial,

[1] 'Making predictions' is vaguely used. An example of a downstream task that would make predictions could be the task of finding answers in a text for question answering.

which makes this process time and resource consuming. This is a major bottleneck for someone wishing to use these pre-trained models to suit their specific needs.

In this project, we target the fine-tuning aspect of these Transformer models on a single GPU server and suggest ways in which their performance can be improved. Specifically, we profile the resource usage patterns in these Transformers and demonstrate what the bottlenecks are by studying the CUDA operations in various Transformer architectures.

We then present techniques to exploit the usage behavior of these models. We introduce our system, FastTran, a caching mechanism which maps training examples to their corresponding intermediate outputs.

More specifically, we perform the following tasks:

- Profile the training phase of various Transformer models and collect GPU resource usage data.
- Cache forward propagation results to decrease time of fine-tuning and compare fine-tuning results with the traditional implementation.
- Study the trade-off between memory usage and batch size in FastTran.

In section 2, we present an introduction to pre-training and fine-tuning in Machine Learning and explain the Transformer architecture. In section 3, we propose a system, FastTran and present the two phases of the project, namely, profiling and caching. In the Evaluation section 4, we demonstrate how FastTran improves the time taken for fine-tuning of Transformer models to a fraction of the original fine-tuning time. In section 5, we discuss various works that target different phases of machine learning workload and improve their performance. Finally, in section 6, we discuss possible future directions to explore.

2 Background

2.1 Pre-training and Fine-tuning

Traditionally, to perform a task such as classification or regression on a dataset, a neural network architecture is designed and trained from scratch. The model parameters (neural network weights) are initialized randomly and the network is fit on the dataset to minimize a loss function using some optimization rule to update the parameters. This approach is viable and recommended if the task can achieve good accuracy with a simple neural network with few hidden layers. Especially if the number of model parameters isn't very large compared to the number of training examples, the designed neural network can work well in such a scenario.

However, such training cannot be applied to perform Natural Language Processing tasks because of the following reasons: NLP tasks require understanding of complex patterns that arise in natural language. It is especially challenging because of word/sentence ambiguity, understanding unstructured data, etc. Thus achieving good accuracy on NLP tasks require training very deep neural network architectures which in turn requires a lot of labeled training data to prevent over-parametrization for the models to converge. Unavailability of a lot of labeled data, but presence of a lot of unlabeled data led to many works which explore semi-supervised techniques that capture underlying semantics and patterns of the data. In this workflow, deep neural networks (such as BERT) with millions of parameters are *pre-trained* using lot of unlabeled data such as wikipedia and Google News. The pre-training of these networks takes a lot of time and uses a lot of resources.

The pre-trained networks can be used in the downstream tasks such as sentence classification, question answering, etc. Instead of learning the model parameters from scratch using random-initialization, the same pre-trained model parameter initialization can be used for the task by *fine-tuning* the network on the target dataset.

Two approaches have been considered to perform fine-tuning:

1. **End-to-end fine-tuning**: In this approach, the entire pre-trained neural network parameters are allowed to change during the fine-tuning stage on the target data.
2. **Fine-tuning only the last layer**: In this approach, all the pre-trained neural network parameters except the last fully connected layer parameters are maintained static. The last layer parameters are allowed to change by the optimization algorithm while fine-tuning on the target data.

The accuracy with the end-to-end fine-tuning approach is usually better than the accuracy with the approach that only updates last fully connected layer of the pre-trained model. On the other hand, end-to-end fine-tuning requires a lot of time and compute resources compared to the second approach.

2.2 Transformer architecture

The transformer architecture is an encoder-decoder model originally designed for machine translation task based on Self-Attention mechanism. The model architecture, unlike prior models, does not rely on recurrence or convolutional layers and uses only the self-attention mechanism to encode the sequential structure of the sentences. As a consequence, the sequential computation

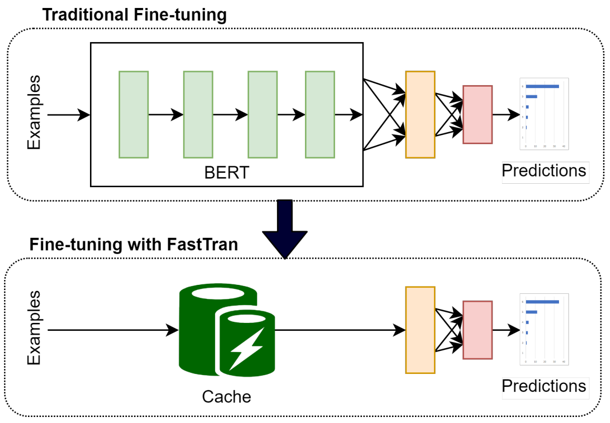


Figure 1: FastTran

doesn't depend on the length of the input sentences and takes constant time. Additionally, by incorporating the word position in the sentence, Transformer layer training can be efficiently parallelized.

The encoding and decoding components of the Transformer consists of 6 encoders and 6 decoders respectively stacked on top of each other. Each encoder module consists of a multi-headed self-attention layer and a feed-forward layer. Matrix multiplications make up most of the calculations that occur in these layers making it the dominant operation that occur in Transformers.

As a result, the Transformer outperforms recurrent and convolutional models in neural machine translation task while requiring less computation to train. Many state-of-the-art models (BERT [3], GPT [23], GPT-2 [25], Transformer-XL [2], XLNet [33], XLM [15], RoBERTa [17], DistilBERT [27], CTRL [11], CamemBERT [20], ALBERT [3]) based on the Transformer architecture have been designed to address diverse array of NLP tasks.

FastTran targets the fine-tuning phase of Transformer based models using a single GPU machine.

3 Design

The design of FastTran consists of two phases: in the first phase, we profile Transformer based architectures to determine the bottlenecks and potential areas of optimization; in the second phase, we employ a caching mechanism to alleviate the run time of Transformer model based on the profiling outcomes.

3.1 Profiling

The first phase of the system consists of profiling GPU usage of existing Transformer model architectures. This helps in identifying the bottleneck, i.e, operations that occupy most of the fine-tuning time. Using these results, we devise a strategy in section 3.2 to improve fine-tuning time. A similar approach can be employed to devise strategies for reducing fine-tuning time for other models.

Three models were profiled: Google's BERT [3], Facebook's XLM [15], and XLNet [33] developed by Carnegie Mellon University and Google.

In the first step, we reduced the complexity of the Transformer model that is to be profiled so that we target the specific things we want to profile. Specifically, we reduced the number of stacked encoders and decoders in the Transformer module to 1, reduced the number of attention heads to 1 and trained only for one batch of data. Doing this will run only one forward propagation and one backward propagation step on the model. We used NVIDIA CUDA Command-Line Profiler `nvprof` to obtain the profiling results and used NVIDIA CUDA Visual Profiler `nvvp` to observe the profiling results in an easy to analyze UI. A screenshot showing the results of `nvvp` profiling is presented in the appendix. The results obtained by `nvprof` gave us insights into the coarse-grained CUDA kernel calls invoked during one step of training a Transformer model. `nvvp` also gives some insights such as optimizing which kernel calls will give most benefit in terms of efficient execution.

To obtain fine-grained profiling results, we used PyTorch's Autograd Profiler. The Autograd Profiler lists GPU CUDA operations and their relative usage percentages during training of each model.

Fine-tuning a pre-trained model of which most of the parameters are static can intuitively benefit from the strategy of caching the final layer outputs regardless of the model architecture. Profiling results for Transformer models show that most of the computations in the forward propagation of the model are matrix multiplications. By caching the final layer outputs of the pre-trained model in the first epoch of fine-tuning and reading them directly from the cache in the subsequent epochs, we can skip all these matrix multiplications, thereby reducing the time taken to fine-tune the model.

3.2 Caching

We now describe the caching mechanism that has been explored in FastTran. As shown in Figure 1, in the traditional fine-tuning setup, the examples are fed into the transformer model to perform a forward propagation, followed by a classification head specific to the fine-tuning task. This process is performed for each batch of input examples in each epoch. Let's denote the output of the last layer of the transformer model as I. Let's consider an input example X_i with label Y_i. Let I_i be the intermediate layer output for X_i. Since the weights of the transformer model are fixed, i.e. we do not perform backpropagation, each time we perform a forward propagation for input X_i, it results in the same I_i. In FastTran, instead of performing the forward propagation in each epoch and re-calculating I_i, we cache this output in GPU during the first epoch. For subsequent epochs, we load the final layer outputs from the cache, re-use these values, and perform the training on the task specific model. With this approach, the model architecture remains the same. Hence, there is no degradation in the accuracy.

However, one might run into some challenges during caching. One of the challenges that we faced while implementing the caching mechanism was limitation of GPU memory. If the available memory on the GPU is smaller than the total size of the cached data, then the caching approach as described in this section cannot be applied directly. We propose a few potential solutions below that could be employed to alleviate this problem. We also discuss various trade-offs that one may need to make to decide which solution to use.

- If the size of last layer outputs for the whole dataset is large and cannot fit in the GPU RAM, instead of caching them on GPU, we can store the results in CPU RAM in the first epoch. In the subsequent epochs, these values can be loaded into the GPU RAM in batches.

- To further improve the performance, since the model is trained by batching the data, these values can be *pre-fetched* into the cache concurrently with previous batch execution for the next batch. This will overlap the training time with the caching time, reducing the total run-time for fine-tuning.

- If the total size of the output is too big to fit even in the CPU RAM, the percentage of outputs cached can be adjusted based on the available RAM. In this case, the forward propagation need be performed only for examples whose outputs are not cached.

- Another approach to handle large cache size is to use some kind of quantization on the cached data. In this approach, lower precision floats can be used to represent the cached data, thereby reducing the amount of space occupied by the cache on the GPU. As a downside of using a lower precision representation, the model's accuracy might reduce. But this approach is faster than the above techniques as it avoids the overhead of accessing information from CPU RAM and transferring it to the GPU RAM which is slower than directly reading data from GPU RAM.

- **Improved sampling:** Since the model is trained batch-wise where the fine-tuning examples are picked randomly, it can lead to a scenario where within a batch, the intermediate layer outputs are available in the cache for some of the examples only. Loading from cache for a subset of examples, and performing the forward propagation through the transformer model for the remaining can reduce the performance. To alleviate this, we propose an alternative sampling approach for fine-tuning. In this approach, one can only compute final layer outputs for a batch of input data, cache it in GPU RAM and perform some number of mini-batch optimization steps only on this batch of data for a few iterations before moving onto the next batch. In this way, the caching mechanism will not be limited by the available GPU memory by choosing an appropriate batch size. However, one has to analyze whether this new approach of performing few optimization steps on same batch before sampling a new batch will lead to a reduction in accuracy.

4 Evaluation

4.1 Experimental Setup

We implemented FastTran on top of the Hugging Face Transformers repository[2]. For performing the experiments, we deployed a VM on Google Cloud Platform consisting of 2 vCPUs with 13 GB memory and 1 NVIDIA Tesla V100 GPU with 16 GB memory.

We ran the experiments on the Question Answering task using SQUAD v1.1 dataset [26]. We split the dataset into two sizes: a small dataset with 3433 training examples and a large dataset with 89632 training examples.

To implement the fine-tuning of the models without caching, we used the `torch.no_grad()` scope for the forward propagation of the BERT model parameters that are to be kept static. We implemented fine-tuning with caching in two steps. In the first step, we perform a forward propagation on all the input examples as done in the first epoch and we store the final layer outputs of the pre-trained model in the cache along with the input examples as a `TensorDataset` in order to reuse the existing sampling and data loading pipeline. In the second step, which is the actual training step, instead of running the forward propagation again and using the existing `TensorDataset`, we use the cached `TensorDataset` stored in the first step. We train the task-specific layers of the network using this cached `TensorDataset`.

[2]https://github.com/akshatabhat/transformers

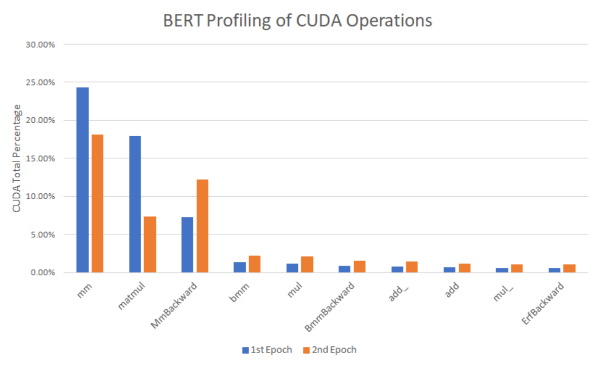


Figure 2: Profiling BERT

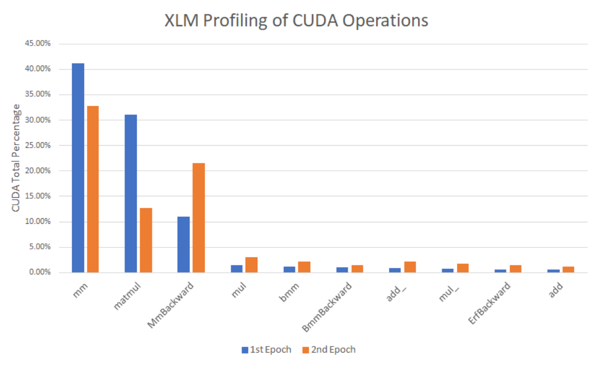


Figure 3: Profiling XLM

4.2 Profiling

We now discuss the results of end-to-end profiling performed using PyTorch Autograd Profiler. Note that the profiling experiments were carried out on the original transformer models without our caching mechanism to determine the bottleneck of the existing architectures. These experiments were performed for one batch and two epochs. For BERT, in Figure 2, we observe that matrix multiplications in the forward propagation (mm and matmul) occupy a large portion of the total CUDA time (42%), followed by backward propagation for these operations (9%). We observe similar pattern for XLM in Figure 3. However, for XLNET in Figure 4, backward propagation for matrix multiplication, Einstein summation, and index select occupy most of the CUDA time (51%).

We observe that the CUDA total Percentange varies from first epoch to second epoch. For example, this difference can be seen in BERT and XLM, where mm and matmul occupy a larger percentage of CUDA time in first epoch (24% and 18%) in comparison to the second (18% and 7%). Conversely, MmBackward seems to occupy a larger percentage of CUDA time in second epoch (7%)in comparison to first epoch (12%). The values observed in the second epoch remain the same in subsequent epochs.

We also use PyTorch Autograd profiler's `record_function()` to label blocks of code performing forward propagation, backward propagation and parameter updates, in order to capture the CUDA time in each of these blocks for BERT. From this, we observe that the total CUDA time occupied by all the operations in the forward propagation is more than the backward propagation in the first epoch. However, in the subsequent epochs, the time spent in the backward propagation is more than the time spent in the forward propagation.

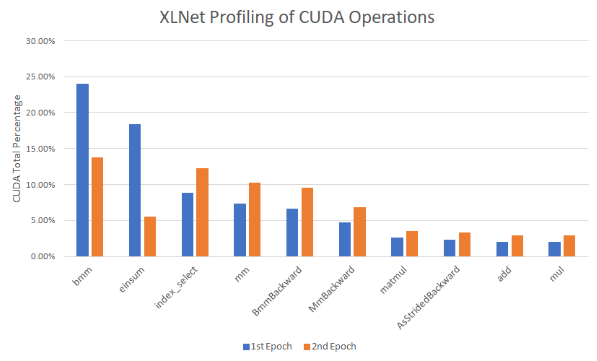


Figure 4: Profiling XLNET

4.3 Caching

In this section, we discuss some results of the caching phase. Specifically, we study the trade-off of batch size vs GPU memory usage, time taken to fine-tune with caching, and the size of cached data in GPU memory.

Table 1 shows the time taken to fine-tune the BERT model as described in section 2.1 in two settings: the conventional approach and the caching approach. In the caching approach, a majority of the fine-tuning time is spent in computing the cache and storing it in the GPU. After caching the final layer outputs in the GPU, the time taken per epoch is less than half a second for the small dataset and about 10 seconds for the large dataset which is very small comparable to the time to cache. As a result, the total time taken to fine-tune with caching is orders of magnitude smaller than the total time taken to fine-tune without caching without any drop in model accuracy. The benefit comes because of the fact that most of the time taken in fine-tuning without caching is in the forward propagation step which involves a lot of matrix multiplications and hence needs a lot of time and compute resources.

In table 2, we analyze the amount of space occupied by the cache in GPU while fine-tuning for different tasks.

Dataset Size	**No Caching**	**Caching**	
	Total Time (s)	*Caching Time (s)*	*Total Time (s)*
Small (5 epochs)	155.43	32.63	**34.72**
Large (2 epochs)	1631.86	871.57	**889.55**

Table 1: Caching time: small dataset: 3433, large dataset: 89632. Batch size: 12. Caching significantly improves total time by bypassing the Transformer forward propagation.

Task	**Last fc layer dim**	**Cache Size**	
		Small Dataset	**Large Dataset**
Sentence Classification	768	10.06 MiB	262.59 MiB
Question Answering	384 x 768	3.77 GiB	98.47 GiB*

Table 2: Number of training examples: small dataset: 3433, large dataset: 89632. Batch size: 12. *Doesn't fit in the GPUs we experimented.

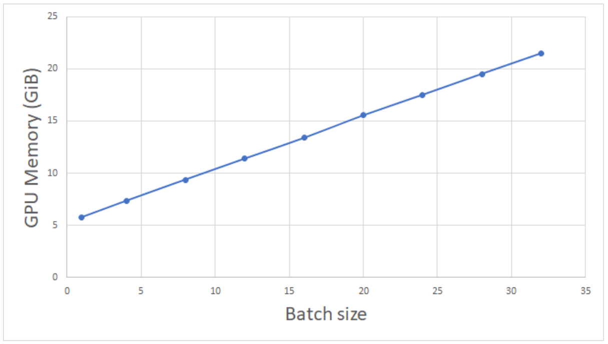


Figure 5: Batch size vs GPU Memory

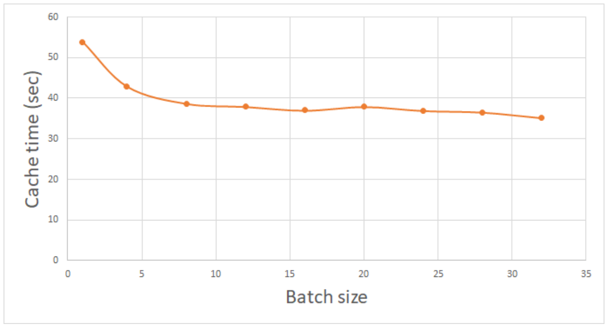


Figure 6: Batch size vs Time to Cache

We experimented with different tasks that require different sizes of the last fully connected layer of the pre-trained model to be cached. The sentence classification task just needs last-layer hidden state of the first token (classification token) of the sentence which is of dimension 768. On the other hand, the question answering task requires last-layer hidden states for all the words in the sequence (shape: 384 x 768). As a result, for the sentence classification task, the cached data even using the large dataset is about a quarter of a Gigabyte. On the other hand, caching for question answering task for small dataset takes up about 4 GiB storage and for the large dataset, the cache doesn't fit in the GPU.

In figures 5 and 6, we analyze the effect of batch size in time taken to cache and the GPU memory consumed. As the size of the batch increases, the time taken to cache decreases as the model can batch the forward propagation computations for multiple training examples at once. On the other hand, as the batch size increases, the total GPU memory consumed increases linearly as seen in figure 5. Therefore, there is a clear trade-off between the time taken to build the cache and the availability of GPU memory.

5 Related Work

Various systems frameworks have been developed to help with the training and serving of machine learning models. Bismarck [4] has been proposed as an in-RDBMS architecture, useful for training smaller models containing fewer parameters but not applicable for Transformer models. The parameter server model [16] works well for distributed training of bigger machine learning models where the number of model parameters and the amount of training data is large, which can be applicable to the large-structured Transformer models.

Several other systems have been proposed to help accelerate Machine Learning training. Orion [30] uses distributed shared memory along with a dependence analysis mechanism to determine if parallelism is effective and schedules distributed computations. Parallax [12] exploits the sparsity of NLP model parameters to optimize data parallel training by combining the Parameter Server [16] and AllReduce architectures to reduce the amount of data transfer. [32] exploits the advent of specialized hardware, Remote Direct Memory Access (RDMA), to reduce the bottleneck of RPCs in ML training. But, none of the above approaches specifically target the workload behavior in Transformer models. LARS [34] suggests a new training approach designed specifically to address large convolutional neural network training, but it performs poorly for attention models such as Transformers. In LAMB [35], using large mini-batches and layerwise adaptive optimization demonstrates that training BERT can be done in just 76 minutes. However, these approaches improving the training time and the problem of inefficient fine-tuning is still left unaddressed.

Various techniques have been explored to compress machine learning models to accelerate the fine-tuning or inference. Quantization is a technique of decreasing the numerical precision of a model's weights. Post-training quantization like k-means clustering quantization has been explored in [7], though such methods don't improve memory requirements or speed. Quantization-aware training has been explored in [10] which can be used to increase performance for predictions. Pruning is another technique that has been explored: by removing individual connection weights as explored in [5, 8], the matrices are made sparser, and implementations like TensorFlow accelerate sparse matrix multiplications. Neuron Pruning, where entire neurons are removed as explored in [22], makes weight matrices physically smaller which in turn makes computations with them faster. Removing entire weight matrices has been explored in [21] where they remove entire attention heads from big transformer-based models with minimal accuracy losses. Knowledge distillation is another technique that has been explored in [13, 18, 27, 28]. After learning a big model (teacher), a smaller model (student) is trained to mimic the teacher's behaviour (either the output or internal data representations). However, all the above techniques involve either making changes to the existing model or require the creation of an additional, simpler model.

Some work has also been done in the context of Transformer models. [6] uses a stacking algorithm to transfer knowledge from a shallow model to deep model and apply stacking to progressively accelerate and achieves similar performance with faster training. In [36], they use masking tricks and dynamic programming to speed-up the decoding phase by 4x, but has comparable training time in total. However, both these techniques improve the training performance by making changes to the model architecture.

Freeze Inference [14] is a system that caches intermediate layer outputs only for efficient deep learning inference. As the workload targeted by Freeze Inference is inference, the system cannot make any assumptions about the new examples it may encounter for inference. As a result, Freeze Inference uses techniques such as computing the similarity of the intermediate layer activations of the new example with the cached data and approximates the prediction of the model based on the similarity. On the other hand, the workload targeted by FastTran is fine-tuning in which the same training data is repeatedly sampled (possibly randomly) in each epoch. FastTran also targets fine-tuning Transformer models in which most of the pre-trained model parameters are kept static. Consequently, FastTran can employ simple caching techniques without having to perform nearest-neighbor computations on the cached weights, while reducing the fine-tuning run-time and achieving the same accuracy as the traditional fine-tuning without caching.

Gandiva [31] is a cluster scheduling framework that exploits the GPU usage pattern in deep learning jobs and improves GPU cluster utilization for deep learning. Themis [19] is another scheduling framework that addresses the problem of unfair GPU allocation during deep learning training at the expense of accuracy. Gandiva and Themis target performance improvement in a cluster setup. Instead, the development of FastTran is solely focused on exploring the trade-off between memory and performance using caching on a single GPU server.

6 Future work

In this project, we profiled the GPU usage of few Transformer based models. To exploit the behavior of resource usage, we then explored a simple caching mechanism to reduce the run-time of fine-tuning the Transformer models. We experimentally evaluated the proposed mechanism in a system called FastTran and demonstrated the system's ability to efficiently fine-tune Transformer models without any reduction in accuracy.

In this project, we only targeted a setup consisting of a server with a single GPU. Even though the techniques proposed in FastTran may apply to a distributed setting with multiple GPUs, it would be interesting to explore the new challenges that emerge. So, a future project could attempt to exploit workload behavior to improve scheduling decisions in Transformer models for efficient utilization in a cluster environment.

FastTran only looks at one approach of performing fine-tuning in which the pre-trained model parameters are kept static. Another interesting venture would be to explore improving the performance of end-to-end fine-tuning approach in which all the pre-trained model weights are allowed to train. The challenges in this setting that need to be addressed are which layers of the pre-trained model to cache, and the trade-offs between available GPU memory, time, and accuracy achieved. Compared to the profiling approach used by FastTran, doing so in an end-to-end fine-tuning setting might require profiling at an even lower level. For example, opening up the Transformer model and profiling each of the individual components such as the encoders, decoders and the attention modules might give further insights into how the caching can be performed.

References

[1] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[2] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[4] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336, New York, NY, USA, 2012. ACM.

[5] T. Gale, E. Elsen, and S. Hooker. The state of sparsity in deep neural networks, 2019.

[6] L. Gong, D. He, Z. Li, T. Qin, L. Wang, and T. Liu. Efficient training of BERT by progressively stacking. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2337–2346, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[7] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[8] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks, 2015.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[10] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[11] N. S. Keskar, B. McCann, L. R. Varshney, C. Xiong, and R. Socher. Ctrl: A conditional transformer language model for controllable generation, 2019.

[12] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun. Parallax: Sparsity-aware data parallel training of deep neural networks, 2018.

[13] Y. Kim and A. M. Rush. Sequence-level knowledge distillation, 2016.

[14] A. Kumar, A. Balasubramanian, S. Venkataraman, and A. Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[15] G. Lample and A. Conneau. Cross-lingual language model pretraining, 2019.

[16] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[17] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[18] P. Luo, Z. Zhu, Z. Liu, X. Wang, and X. Tang. Face model compression by distilling knowledge from neurons. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 3560–3566. AAAI Press, 2016.

[19] K. Mahajan, A. Singhvi, A. Balasubramanian, V. Batra, S. T. Chavali, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient gpu cluster scheduling for machine learning workloads, 2019.

[20] L. Martin, B. Muller, P. J. O. Suárez, Y. Dupont, L. Romary, Éric Villemonte de la Clergerie, D. Seddah, and B. Sagot. Camembert: a tasty french language model, 2019.

[21] P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one?, 2019.

[22] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference, 2016.

[23] A. Radford. Improving language understanding by generative pre-training. 2018.

[24] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[26] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

[27] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019.

[28] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin. Distilling task-specific knowledge from bert into simple neural networks, 2019.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017.

[30] J. Wei, G. A. Gibson, P. B. Gibbons, and E. P. Xing. Automating dependence-aware parallelization of machine learning training on distributed shared memory. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 42:1–42:17, New York, NY, USA, 2019. ACM.

[31] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, Oct. 2018. USENIX Association.

[32] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 44:1–44:14, New York, NY, USA, 2019. ACM.

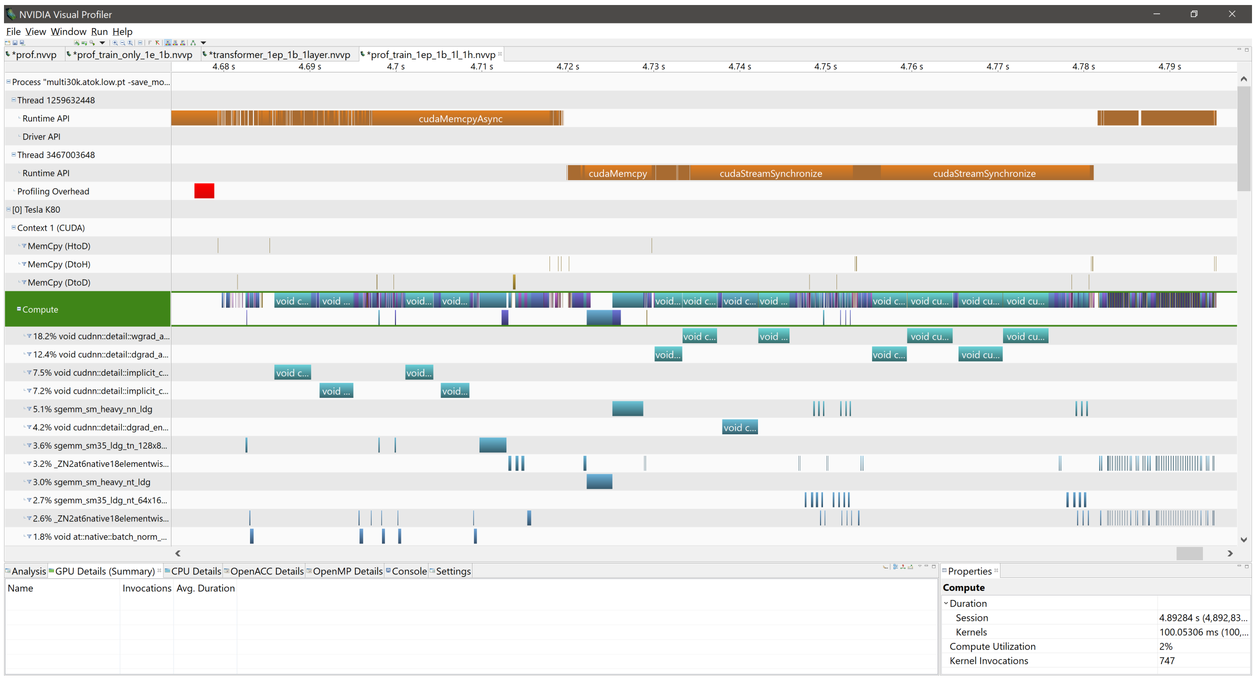
[33] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2019.

[34] Y. You, I. Gitman, and B. Ginsburg. Scaling SGD batch size to 32k for imagenet training. *CoRR*, abs/1708.03888, 2017.

[35] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes, 2019.

[36] B. Zhang, D. Xiong, and J. Su. Accelerating neural transformer via an average attention network. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1789–1798, Melbourne, Australia, July 2018. Association for Computational Linguistics.

A Appendix


NVIDIA Visual Profiler
File View Window Run Help
*prof.nvvp
*prof_train_only_1e_1b.nvvp
*transformer_1ep_1b_1layer.nvvp
*prof_train_1ep_1b_1l_1h.nvvp
4.68 s
4.69 s
4.7 s
4.71 s
4.72 s
4.73 s
4.74 s
4.75 s
4.76 s
4.77 s
4.78 s
4.79 s
Process "multi30k.atok.low.pt -save_mo...
Thread 1259632448
Runtime API
cudaMemcpyAsync
Driver API
Thread 3467003648
Runtime API
cudaMemcpy
cudaStreamSynchronize
cudaStreamSynchronize
Profiling Overhead
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
18.2% void cudnn::detail::wgrad_a...
12.4% void cudnn::detail::dgrad_a...
7.5% void cudnn::detail::implicit_c...
7.2% void cudnn::detail::implicit_c...
5.1% sgemm_sm_heavy_nn_ldg
4.2% void cudnn::detail::dgrad_en...
3.6% sgemm_sm35_ldg_tn_128x8...
3.2% _ZN2at6native18elementwis...
3.0% sgemm_sm_heavy_nt_ldg
2.7% sgemm_sm35_ldg_nt_64x16...
2.6% _ZN2at6native18elementwis...
1.8% void at::native::batch_norm_...
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
OpenMP Details
Console
Settings
Name
Invocations
Avg. Duration
Properties

Compute	
Duration	
Session	4.89284 s (4,892,83...
Kernels	100.05306 ms (100,...
Compute Utilization	2%
Kernel Invocations	747

Figure 7: Transformer NVVP Profiling Results

A.1 Profiling Results

Here, we present the results obtained for Transformer profiling. We profiled a Transformer model for 1 batch of execution with one layer of encoder and decoder and one attention head. Figure 7 shows a screenshot of the `nvvp` program obtained by profiling the Transformer model using `nvprof`.