AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-tuning

Yuhan Liu, Saurabh Agarwal, Shivaram Venkataraman
University of Wisconsin-Madison

Abstract

With the rapid adoption of machine learning (ML), a number of domains now use the approach of fine tuning models which were pre-trained on a large corpus of data. However, our experiments show that even fine-tuning on models like BERT can take many hours even when using modern accelerators like GPUs. While prior work proposes limiting the number of layers that are fine-tuned, e.g., freezing all layers but the last layer, we find that such static approaches lead to reduced accuracy. We propose, AutoFreeze, a system that uses an adaptive approach to choose which layers are trained and show how this can accelerate model fine-tuning while preserving accuracy. We also develop mechanisms to enable efficient caching of intermediate activations which can reduce the forward computation time when performing fine-tuning. We extend AutoFreeze to perform distributed fine-tuning and design two execution modes that minimize cost and running time respectively. Our evaluation on ten NLP tasks shows that AutoFreeze, with caching enabled, can improve fine-tuning on a single GPU by up to $2.55\times$. On a 64 GPU cluster, for fine-tuning on the AG's news dataset, AutoFreeze is able to achieve up to $4.38\times$ speedup when optimizing for end-to-end training time and $5.03\times$ reduction in total cost when optimizing for efficiency, without affecting model accuracy.

1   Introduction

Deep Learning based models have been shown to provide extremely competitive performance across a wide range of tasks. However, building deep learning based models for new tasks requires a large amount of data and compute [48]. To circumvent these requirements practitioners bootstrap new tasks from existing models. To bootstrap from existing models, practitioners typically use transfer learning or fine tuning [71]. In case of transfer-learning, the features from a large pre-trained model are directly used on a new task and *only the last one or few* layers are trained to develop a specialized model. Closely related is the practice of fine tuning where weights from a pre-trained model are used to initialise a new task; following initialization *all layers* of the model are trained until convergence.

In the case of language models [18], fine-tuning has become a standard part of the two stage training process. In first stage, which is pre-training, complex models (e.g., BERT [10]) are trained with a large corpus of unlabeled data. In the second stage, which is fine tuning, the pre-trained model is fine-tuned for a specific task such as sentiment analysis [32] or topic classification [69] etc.

While fine-tuning is cheaper than pre-training a model, it is important to note that pre-training is usually performed very infrequently compared to fine tuning. For example in the case of language models, a vast majority of practitioners take a pre-trained BERT model and perform fine tuning on their data sets [32, 46, 67, 16, 69, 55] or for new tasks [55, 60, 70, 28].

Prior approaches in developing tools for improving ML model training exploit ways to reduce training time by maximizing throughput and better utilizing resources. A number of works including PyTorch DistributedDataParallel (DDP) [25] and BytePS [21] provide support for data-parallel distributed training and improve utilization of heterogeneous resources such as GPUs, CPUs, and network bandwidth. More recent works like Cerebro [39] and PipeDream [40] utilize hybrid parallelism to improve model selection throughput and minimize synchronization bottlenecks.

Even when using Pytorch DDP [25], we find that fine-tuning BERT for the sentiment classification task
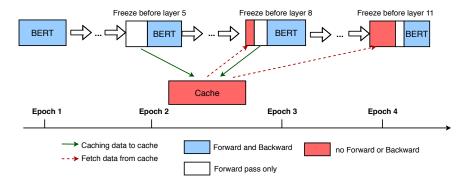
Figure 1: We present a high level design of AutoFreeze. During fine-tuning, AutoFreeze adaptively determines layers which can be frozen. Once layers are frozen, the backward computation for those layers can be avoided. At later epochs, intermediate outputs are also cached leading to further gains.

with the Yelp dataset [69] can take around 27 hours [1] in a four P100 GPU cluster.

The primary reason fine tuning is so computationally expensive is the large size of the pre-trained model. Each layer of the pre-trained model is computationally intensive (Section 2.2) and requires a significant amount of time for forward and backward pass. Further, existing training systems are generic and hence oblivious to the convergence properties of the fine-tuning workloads; given that fine tuning already starts from a pre-trained model, training systems need to be aware of the opportunities that arise from rapid convergence to achieve high throughput.

A natural approach to improve fine-tuning performance is to limit the number of layers of the model that are updated, thus making it similar to transfer learning. For example, if we consider $BERT_{BASE}$ which has 12 encoding blocks, prior approach by Lee et al. [24] trains a fixed number of blocks (e.g., the last 3 blocks) and *freezes* the weights for the remaining blocks. However, this approach can affect the final model accuracy. For example with the MRPC dataset [11] this approach can reduce the time for an epoch by $2\times$, but we find that the accuracy of the fine-tuned model suffers, dropping from 87% to 76.5%.

Another approach used in prior work by Chen et al. [6] is to apply the "Lottery Ticket Hypothesis" to identify matching subnetworks in pre-trained BERT models to enforce sparsity in models trained for different downstream tasks. While this approach retains accuracy and can lead to sparser models, it does not lead to improvements in training speed without dedicated hardware or libraries [30].

In this paper, we propose a novel approach where the number of model blocks that are updated and resources used during fine-tuning are adaptively chosen during the fine-tuning process. Our work is inspired by recent work of Raghu et al. [45] who developed SVCCA, a new metric that captures how different layers of model change over the course of training. The SVCCA score for a layer, as proposed by Raghu et al. [45], is computed by comparing the intermediate model weights with the final weights and can thus be used for post-hoc analysis. Applying that approach to model fine-tuning we observe that the initial layers of the model converge rapidly and thus we can *freeze* such layers. Freezing the initial layers means that the backward pass for those layers can be skipped, thereby reducing the computation and communication required. While SVCCA scores show the promise of freezing layers early, we still need an online algorithm that can decide which layers should be frozen and when. We develop a gradient-norm based test that ranks layers by their rate of change and based on it, selects the slowest changing layers for freezing. We show that our method is effective at detecting when layers should be frozen without affecting accuracy across multiple datasets.

Beyond just reducing the time for backward pass freezing early layers of the model provided several advantages. One major advantage is that freezing layers will also reduce the amount of communication required for distributed training. For example freezing layers of BERT can save around 27MB per layer frozen [2], thus reducing the synchronization required.

Another potential benefit of freezing is that it can reduce the time for forward pass. Since the frozen layers don't change during subsequent training iterations, the output for a given data point will be constant

---

[1]Measured on CloudLab
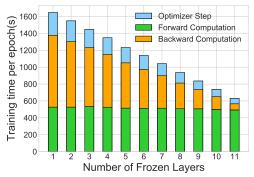[2]Size of one BERT encoder layer.

Figure 2: **Potential Benefits of Freezing:** For IMDb dataset we show the potential savings in time when performing freezing. Time for forward pass is constant since we need to perform a full forward pass before calculating gradients. On the other hand timing for backward pass reduces as we increase number of layers frozen.
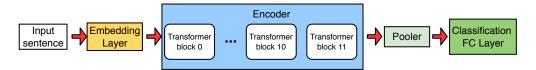

Figure 3: **BERT model architecture**

for the frozen layers. Applying this insight we can cache the output of the forward pass up to the layer that has been frozen. Once the same data point is selected to be used again for training, we can load the pre-computed intermediate values from our cache and continue training. Figure 2 shows the potential saving our freezing module can provide by caching the intermediate outputs.

We design AutoFreeze, a system for automatically freezing layers to accelerate fine-tuning. Our system consists of two main modules on each GPU: a *freezing module* that has a pluggable decision engine that can make decisions on which layers should be frozen based on the aggregated gradients as training progresses. The freezing module also includes a decision engine that dynamically picks the amount of resources to use during fine-tuning. We design two modes that the user can choose from: Performance Packing for minimizing training time or Efficiency Packing for minimizing cost. We also design a *storage manager* module to implement the caching functions described above and the storage manager handles a number of common concerns in caching, including selecting the appropriate backend (CPU memory / SSD etc.) and deciding when to evict data from the cache.

We evaluate AutoFreeze using a wide range of fine-tuning tasks. including topic classification on the AG's News dataset [69] and Sogou News dataset[55], sentiment analysis on Yelp Full dataset[69] and IMDb dataset[32], question answering on SQuAD2.0 dataset[46], multiple choice task on SWAG dataset[67], and text summarization on CNN/DailyMail dataset [16]. We find that for a single machine fine tuning AutoFreeze can improve training time by up to $2.55\times$ while affecting accuracy by less than 0.1%. We also show that AutoFreeze is especially effective for large datasets like Yelp where freezing layers reduces fine-tuning time from 52.5 hours to 27 hours and caching further reduces this to 24.6 hours. In distributed fine-tuning AutoFreeze can significantly improve training time and efficiency, reducing the end-to-end training time by $4.4\times$ with the performance packing mode and reducing total cost by $5.03\times$ when using the efficiency packing mode for fine tuning on the AG's News dataset in a 64 GPU cluster.

2 Motivation and Background

In this section we first provide background on model fine-tuning and transfer learning and detail why fine-tuning is expensive. Following that we motivate how freezing or limiting the number of layers of a model trained can lead to significant savings. Finally we show how static schemes that freeze a constant number of layers are ineffective.

2.1 Model Fine-tuning

Transfer learning and fine tuning of large pre-trained models has enabled easy use of deep models for new tasks and new datasets [55, 60, 70, 28]. In transfer learning we use the features from a large pre-trained model and train only the last few layers to specialize on a new task, while in case of fine tuning the whole pre-trained model is trained on a new task. Both transfer learning and fine tuning have several advantages over training models from initialization including i) enabling use of deep models when training data is scarce, ii) transferring common features among related tasks iii) significantly faster convergence that also reduces the computation time.

Further, periodic fine-tuning is a necessity for models deployed in production, since in real world the data distribution changes quite frequently [57]. When data distribution drifts further from the training distribution, models typically observe increased error rates from out of distribution(OOD) data points [26, 34]. Periodically fine-tuning models with newly collected data is one of the most common methods to keep error rates low [57, 26, 72], making fine-tuning an extremely important workload for ML deployments in production.

Although feature based transfer learning is very popular in computer vision tasks [52, 19, 3, 63, 5], recent works [18, 43] show that language models enjoy significantly better performance when using fine tuning. However, even when performing fine tuning, large models like BERT [10] require a significant amount of time. For example, fine tuning BERT on the relatively small IMDB dataset [32], containing 25K points, takes around 3 hours on a single P100 GPU. On larger datasets like Yelp (Table 2) we see that fine-tuning can take more than two days on single P100 GPU. Distributed training provides limited benefits given the size of the models [47], with four P100 GPUs only reducing the training time to 27 hours. Even on the latest A100 GPU, fine-tuning $BERT_{LARGE}$ on the Yelp dataset can take around two hours. Thus, the exorbitant cost of fine tuning becomes a limiting factor for data scientists in developing new models.

2.2 BERT Model Architecture and Timing

To get a deeper understanding of the performance of fine-tuning on BERT, we first discuss the model architecture and then present a breakdown of fine-tuning time. We primarily focus on $BERT_{BASE}$ which is depicted in Figure 3. $BERT_{BASE}$ has 12 encoder layers, which are also called Transformer blocks. Each Transformer block is identical and comprises of a self-attention layer with 12 attention heads and a fully connected layer of size 768. In this work we refer to layer and transformer block interchangeably, therefore by freezing a layer we mean freezing the entire transformer block.

As discussed in prior work [55], fine-tuning for text classification tasks typically only requires around 4 epochs to achieve state-of-the-art accuracy. But the time taken per epoch is high because of the following reasons:

Memory constraints: With the BERT model being large (model weights around 420 MB) and the intermediate activations of each layer also being large, the batch size that can be used for fine tuning is limited by GPU memory. With an NVIDIA P100 GPU, having 12GB of memory, we observe that we are limited to a batch size of 6 [3]. Even on newer hardware like A100 GPU having 40GB of memory, the maximum batch size that can be supported is limited to 32.

Computation needs: The transformer blocks discussed above are also compute intensive and we observe that the gradient calculation time, especially in the backward pass can be significant. For example, when fine-tuning with the IMDb dataset, we see that doing one iteration takes around 435ms of which more than 50% is taken by the backward pass.

Communication intensive: The large model size also imposes significant communication overheads when performing distributed data-parallel training. For example, when scaling from one p3.2xlarge to 64 p3.2xlarge GPUs on Amazon EC2, while having a fixed batch size per GPU, leads to an inflation in per-iteration time from 0.29 seconds to 8.9 seconds.

---

[3]Measured using PyTorch 1.0.1

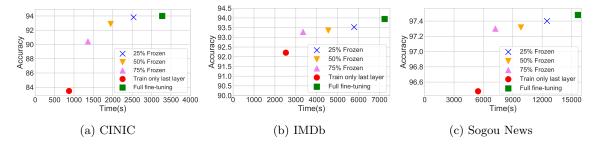(a) CINIC   (b) IMDb   (c) Sogou News

Figure 4: **Evaluating Static Freezing:** We evaluate performance of static freezing (freezing a fixed number of layers) on pre-trained ResNet-18 and BERT. In (a), we use ResNet-18 pre-trained on CINIC-10 for fine-tuning on CIFAR-10. In (b) and (c), we fine-tune BERT$_{BASE}$ on IMDb and Sogou News datasets. Static schemes improve fine-tuning time but often lead to loss in accuracy.

2.3 Existing Approaches

One direct approach to reduce the computational cost is to only fine-tune a subset of the layers [24]. For example, as shown in Figure 2, only updating the last k layers of the BERT leads to an almost linear decrease in time per iteration. This hints that avoiding gradient computation for certain layers, i.e., *freezing*, can significantly reduce training time.

We note that in order to realize the gain from freezing, the layers should be frozen in order, *e.g.* freezing layers 3 and 4 before freezing earlier layers 1 and 2 is not going to provide any speedup. This is due to the usage of automatic differentiation [33] in popular deep learning libraries for backpropagation. Automatic differentiation uses the gradient of later layers to calculate the gradients of earlier layers, *i.e.*, to calculate gradients of Layer 1 automatic differentiation requires gradient of Layer 2 to be calculated. However, this leads to the question of which layers should be frozen and how does freezing of layers effect the model accuracy.

First, we consider simple static freezing schemes where a fixed number of layers are chosen to be updated during training as presented in [24]. In Figure 4 we compare static freezing schemes when fine-tuning $BERT_{BASE}$ with IMDb and Sogou dataset. We compare training the last 25%, 50%, 75% of the layers, or only the last layer, to full fine-tuning. We see that such static freezing schemes lead to 0.5% to 1.7% accuracy drop for IMDb. On the other hand for Sogou, we observe that while some static freezing schemes only suffer 0.2% accuracy loss, training only the last layer still leads to significant accuracy loss. We also see similar results for an image classification workload where we fine-tune ResNet-18 model [15] pre-trained on CINIC-10 [9] with CIFAR-10 dataset. In Figure 4a we again see that training only the last layer leads to around 10% reduction in accuracy, while training 50% of the layers leads to 1.12% reduction in accuracy.

Other approaches to improve fine-tuning [6] use the "Lottery Ticket Hypothesis" to identify matching subnetworks such that a specific sparsity can be achieved. However, finding the sparsified subnetwork does not provide direct speedup because these approaches use unstructured pruning that prunes individual weights. EarlyBERT [7] on the other hand identifies lottery-tickets in the early stage of BERT training, which provides training speedup compared to baselines, but results in accuracy degradation.

Overall, our results show that existing schemes either affect the accuracy of fine-tuning or provide limited speedups on existing hardware. Next, we propose developing a novel adaptive method that can select appropriate layers for freezing, thereby improving performance without affecting accuracy.

3 AutoFreeze Design

We next describe the design of AutoFreeze, a system for automatically freezing model layers during fine tuning. We first discuss the scheme used by AutoFreeze to decide which layers to freeze and following that discuss how AutoFreeze can automatically cache intermediate outputs to further improve performance. Finally, we describe how AutoFreeze can improve resource utilization in distributed settings.
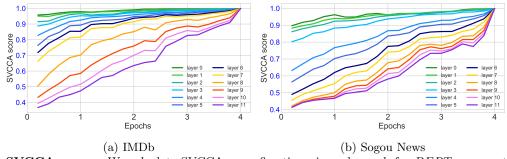
| (a) IMDb | (b) Sogou News |

Figure 5: **SVCCA scores:** We calculate SVCCA score five times in each epoch for $BERT_{BASE}$ on two datasets (a) IMDb (b) Sogou. This shows that layers of a neural network converge bottom-up, *i.e* earlier layers converge faster than later layers. This bottom-up convergence allows us to freeze earlier layers first allowing us to obtain computational benefits from freezing.

Algorithm 1 Freezing Module

Input: List of layers that are not frozen $activeLayers$, Percentile for freezing N
Input: accumulated gradients for current interval Δ_{T_l} and previous interval Δ_{T-1_l}
for $layer_l$ in $activeLayers$ **do**

$\quad \eta_l = \left| \left\| \Delta_{T-1_l} \right\| - \left\| \Delta_{T_l} \right\| \right| / \left\| \Delta_{T-1_l} \right\|$

end for
for $layer_l$ in $activeLayers$ **do**
$\quad$ **if** $\eta_l < N^{th}$ percentile(η) **then**
$\quad\quad$ freeze $layer_l$
$\quad$ **else**
$\quad\quad$ **break**
$\quad$ **end if**
end for

3.1 Adaptive Freezing for Fine-tuning

As described in Section 2.3, statically determining which layers to freeze can lead to reduced accuracy. Our insight is that layers which are closer to convergence are good candidates for freezing and by periodically inspecting the progress of each layer we can determine when a layer can be safely frozen.

We validate our intuition by using SVCCA [45], a recently proposed metric for understanding convergence of neural networks. The SVCCA score is a metric which evaluates the similarity between two layers of neural network. To understand convergence of each layer individually Raghu et al. [45] perform a post-hoc analysis. They calculate SVCCA score by comparing the layers of the model during training with the layers of the already converged model. We use the same IMDB dataset as before and compute the SVCCA scores by comparing each layer's weights periodically (5 times every epoch in this case) with the final weights of the model. SVCCA scores range from 0 to 1 with 1 indicating an exact match (i.e., that the intermediate weights match the final model weights). The results of this experiment are depicted in Figure 5. We observe two main takeaways from this experiment in Figure 5. First, we see that layers of the model converge in order with earlier layers (e.g. layers 0-4) reaching high SVCCA scores within one epoch. Second, while some layers converge fast, others take significantly long time. This indicates that an adaptive freezing scheme can provide performance benefits by freezing layers as they converge.

The above data validates our intuition about the benefits of adaptively freezing model layers. It also shows that SVCCA score will be an ideal metric for freezing since it can track convergence of a layer and freeze it once the layer reaches convergence. However calculating the SVCCA scores shown in Figure 5 requires knowledge of the final model weights, making it inapplicable in practice. Thus we need an online method that can estimate if a layer can be frozen without knowing the final model weights. We next describe how we can use the gradient values at each layer to estimate this.
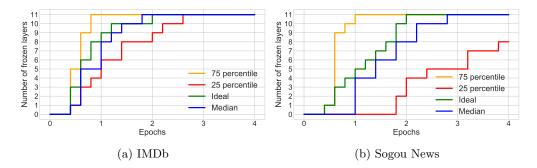
(a) IMDb   (b) Sogou News

Figure 6: **Comparing Gradinet Norm Test with Ideal:** We compare our gradient norm test for $\eta_l = 25$, 50 and 75 percentile, with an ideal SVCCA score based scheme on two datasets (a) IMDb (b) Sogou. We define the ideal scheme as freezing layers with SVCCA scores over 0.9 at each evaluation interval. We observe that gradient norm test with $\eta_l = 50$ percentile (median) closely matches the ideal scheme.

3.1.1 Gradient Norm Test

We next present an online test to determine if a layer should be frozen. Our intuition in designing this test is that the rate of change of the gradient values for a layer can be used to determine how fast the model weights are being updated for a particular layer. Consider that we accumulate gradients for each layer in the model (Δ) and perform our test at fixed intervals (T). Then we define the gradient norm change for layer l, η_l, as

$$\eta_l = \left| \left\| \Delta_{T-1_l} \right\| - \left\| \Delta_{T_l} \right\| \right| / \left\| \Delta_{T-1_l} \right\| \tag{1}$$

We next rank the layers in the order of η_l to determine the layer that is changing slowest. Given our earlier observation about how layers converge in order, we can designate a layer to be frozen if all the layers preceding it are frozen and it is the slowest changing layer. However, this assumes a strict order in the rate of change of gradient norms and we can thus further relax this by designating a layer to be frozen if all the layers preceding it are frozen and if its rate of change is in the bottom N^{th} percentile, where N is a tunable parameter. Algorithm 1 describes the above procedure.

Comparison of Gradient Norm Test to SVCCA score In Figure 6, we evaluate the performance of our proposed gradient norm test by comparing it with the ideal SVCCA score based freezing scheme that has access to the final model weights. In the ideal scheme, we denote a layer as frozen if its SVCCA score compared to the final model weights is above a fixed threshold of 0.9. In Figures 6a and 6b, we vary the percentile value used in Algorithm 1 and see that using too low a percentile value (e.g., 25th percentile) can make the test too conservative resulting in fewer frozen layers compared to the ideal. We also see that using too high a percentile value can lead to the test being too aggressive resulting in loss of accuracy. Finally we see that the median closely tracks the ideal freezing scheme. We perform further evaluation of the effect of varying N in Section 4.3.

3.2 Caching Frozen Layers

Freezing a prefix of the model layers can help us avoid running the backward pass on those layers while fine-tuning. However, given that the layer weights are fixed once they are frozen, we can also avoid the forward pass if we are able to materialize and cache the intermediate output in CPU memory/disk. For example, consider a case where 50% of the model layers are frozen after the first epoch. In this case if we can materialize the output of applying the first 50% of the layers and save it to disk, then for the following epochs we can directly load this intermediate data and thus also avoid the corresponding 50% of the forward pass.

However, there are two main considerations in implementing the caching functionality. First, as model intermediate outputs can be large and can take some time for reading data from the cache, therefore we should only use caching when it will be faster than performing the forward pass. Second, the adaptive freezing algorithm described above, the number of layers frozen could be updated within an epoch making it challenging to determine which outputs should be saved and when.

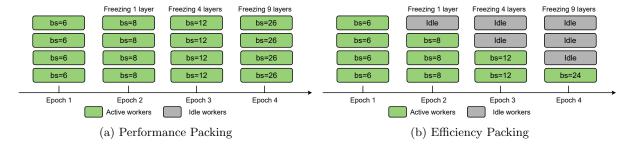(a) Performance Packing

(b) Efficiency Packing

Figure 7: Two modes for distributed fine-tuning enabled by AutoFreeze: (a) Performance Packing: We keep number of GPUs during training constant, and as memory consumption decreases due to freezing we increase per GPU batch size to the max batch size which fits on a single GPU. (b) Efficiency Packing: We reduce the number of GPUs during training to the minimum number of GPUs that can maintain the original total effective batch size (e.g. 24).
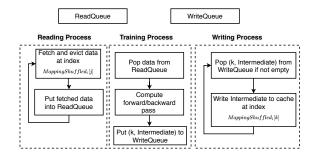

Figure 8: **Storage Manager Design:** In the Storage Manager, reading and writing processes run in parallel to the main training process. The reading process fetches data from cache, while the main training process consumes data from the read queue and produces intermediate outputs. The writing process saves intermediate outputs to cache.

To solve the first consideration, we measure the size and time taken for reading intermediate outputs when fine-tuning the Yelp dataset. For every example, the intermediate output is around 1.57MB and this remains the same across all layers as all the transformer blocks in BERT have the same output size.

However, given that we are limited to small batch sizes (around 6 examples on a P100), we only need to read around 10MB of data for one iteration and this takes around 25ms when using an SSD. On the other hand, doing a forward pass of **one layer** of $BERT_{BASE}$ takes around **11ms**. Thus, in this case, we can see that loading data from SSD should provide a speed-up when more than 2 layers are frozen. In general, the trade-off between caching and repeating the forward pass depends on the disk bandwidth, batch size and computation speed of the GPU. Evaluating this trade-off is not expensive in practice as few iterations of training can indicate how many layers of a model need to be frozen before caching becomes advantageous.

To resolve the second consideration about which layers to store, we design the storage manager which we describe next.

Storage Manager To handle caching of layers we design a storage manager (Figure 8). The storage manager is responsible for managing where data is cached and up to what layer should the forward pass be executed before saving to cache. The storage manager notes down the layer L up to which the model was frozen. In that epoch for all data points that are processed, the output of the forward pass up to layer L is written to cache.

We store the intermediate output to disk when it no longer fits in CPU memory. When the dataset (D points) is larger than the disk space available, we save I points to disk ($I < D$ and I is the maximum number of points that fit on disk). During the next epoch, if the number of layers currently frozen is greater than the number of layers of forward pass that were completed before data was saved to cache, then the storage manager also evicts the data points once they are read. Based on the fact that the number of data points to
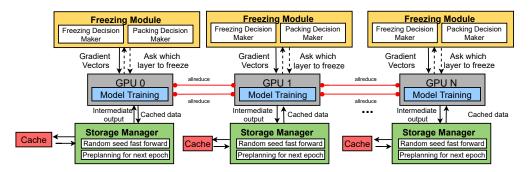
Figure 9: **AutoFreeze System Design:** In the distributed fine-tuning setting, each worker consists of: (1) Freezing Module that accumulates gradient vectors that are aggregated and decides which layers to freeze based on gradient norm test. (2) Storage Manager caches intermediate activations to disk when the overhead of caching is smaller than forward computation. It also fetches data from disk to save forward pass computation for later epochs.

be written will never surpass the number of data points read from disk, we will never exceed the disk space.

Finally as model training typically shuffles data across epochs, the storage manager also ensures that cached data points can be transparently accessed even if only part of the dataset has been cached. We do this by maintaining $MappingShuffled_i$ a mapping from the shuffled indices to the original indices for each epoch i. When writing the intermediate outputs to disk at epoch i, we write at the original indices retrieved from $MappingShuffled_i$. To read the data at index k at epoch $i + 1$ required for training, we read $MappingShuffled_{i+1}[k]$. While this approach incurs random reads/writes to the cache, since each data item is relatively large ($\sim$1.5MB) we have not found this to be an issue in practice. Finally, as shown in Figure 8, we perform read, write and gradient computation in separate processes thereby pipelining I/O with compute.

3.3 Distributed Fine-tuning

We next extend our design to consider how AutoFreeze can help when multiple workers are used for fine-tuning. A common approach to speeding up ML model training (and correspondingly fine-tuning) is to use multiple workers in parallel. In this scenario, the most widely used training mode is a "data-parallel" mode where each worker calculates the gradient on a batch of data and the gradient values are then aggregated using AllReduce to compute the updated model. Adaptively freezing the layers of a model that are being fine-tuned can lead performance and cost-efficiency improvements in a distributed setup. To understand how, we first construct a performance model that captures the computation and communication savings from freezing layers and then explain two distributed execution modes in AutoFreeze.

We consider a distributed training scenario that is similar to DistributedDataParallel (DDP) in Pytorch [25]. We associate T_{comp} as the time to compute gradients on each worker. Assuming that a model can be partitioned into k buckets, where the first $k-1$ buckets are of size b while the last bucket is of size $\hat{b}$, we denote the time for communicating k gradient buckets across p machines using a bandwidth of BW as T_{comm}. In DDP training, the gradient communication is overlapped with the gradient communication and gradients are aggregated (e.g., by calling AllReduce) whenever a bucket becomes ready. Accordingly, using the standard communication model [50, 58], the communication time for a single iteration can be modeled as:

$$T_{comm}(k, b, p, BW) = k \times (\alpha \times (p - 1) + 2 \times b \times \frac{(p - 1)}{p \times BW})$$

where α represents the latency cost (i.e., cost per message sent). The overall time taken to run one epoch for a dataset consisting of N examples with batch size BS becomes the number of iterations times the time per iteration.

$$\frac{N}{BS} \times (max(T_{comp}, T_{comm}))$$

9

In addition to computation savings describe before, in the distributed fine-tuning setup, AutoFreeze provides the following additional savings:

Communication Savings As explained before, our approach in AutoFreeze can lead to skipping gradient computation for layers. Correspondingly we only need to communicate the gradients that have been computed leading to a reduction in the number of bytes sent.

Memory Savings The peak memory used during is dominated by following three components (i) model weights, (ii) gradients and intermediate activations (iii) number of data points (batch size) in the active mini-batch.

By freezing layers AutoFreeze skips the gradient computation of frozen layers thus reducing memory used for storing gradients and intermediate activations. This reduction in memory can be used to increase the batch size or reduce the number workers to improve performance and efficiency as we describe next.

Based on the above savings, we next design two modes that can maximize the efficiency and performance of distributed fine-tuning when freezing model layers: (1) *Efficiency Packing mode* (2) *Performance Packing mode.*

Efficiency Packing mode Our goal in the efficiency packing mode is to *minimize the cost* of distributed fine-tuning. Reducing the cost implies reducing the number of worker-hours used for fine-tuning. Because the time per iteration increases with the number of machines used (latency term α in the performance model), the *most cost efficient configuration is to use the least number of workers* for a given batch size. Thus, in the efficiency packing mode, we increase the per-worker batch size when layers are frozen and correspondingly reduce the number of workers used in training to maintain the total batch size constant as shown in Figure 7b. Denoting the initial total effective batch size as b and initial batch size per worker as b_0, when freezing l layers reduces memory required, we can increase the batch size in a worker to b_l without violating memory constraint. Correspondingly we reduce the number of workers by $\frac{b_l}{b_0}$, thus maintaining the total effective batch size b.

Performance Packing mode In contrast to Efficiency Packing mode, the goal of Performance packing is to *minimize wall clock time* for fine tuning.

From our performance model we observe that for a given cluster of p workers, the wall clock time is minimized when we reduce the number of iterations per epoch. Reducing number of iterations, reduces number of gradient synchronization steps which in turn reduces the total communication overhead. The memory savings from freezing model layers allows us to use larger batch sizes, thus reducing the number of iterations in a single epoch.

More concretely, in the Performance Packing mode, we keep the number of workers used in training p constant and increase the per worker batch size b_i as memory becomes available (Figure 7a) because of freezing layers. Thus, the total effective batch size is increased, reducing frequency of gradient synchronization while using the maximum parallelism available, and thus minimizing the end-to-end training time.

Example: Consider a dataset that contains 120K data points being fine tuned on 64 GPUs. Initially we start with batch size of 384 (6 on each GPU) with no layers frozen. If AutoFreeze decides to freeze the first 11 BERT layers at some point, Performance Packing increases the batch size to 3456 (54 each machine), while Efficiency Packing keeps the total effective batch size as 384 and reduces number of GPU machines to 8 keeping the batch size 384 (48 per machine). For the next epoch after freezing, the time taken with Performance Packing and Efficiency Packing comes out to 36 seconds and 131 seconds respectively. Assuming that cost/second of a GPU is c, the total cost of this epoch with Performance Packing is $36 \times 64 \times c = 2304 \times c$, while that of Efficiency Packing is $131 \times 8 \times c = 1048 \times c$. We can see that Efficiency Packing has $2.2\times$ lower cost but Performance Packing is $3.3\times$ faster.

It is easy for users of AutoFreeze to configure which mode to choose based on their needs. Finally, utilizing Efficiency Packing mode maintains a fixed total effective batch size which we show in Section 4.4 ensures minimum accuracy loss. While using Performance Packing mode users can also configure a maximum batch size and AutoFreeze will ensure that total effective batch size does not go beyond the configured threshold.

3.4 Overall Design, Implementation

Putting the above subsections together, the design of our system, AutoFreeze, is shown in Figure 9. There are two modules on every GPU (e.g. GPU0 in Figure 9): (1) Freezing Module that makes decision on the set of layers to freeze at different intervals of the fine-tuning procedure. (2) Storage Manager that caches

	Full fine-tuning		Frozen up to 9th			Frozen up to 12th			AutoFreeze			Pruning		
	Acc/ Corr	Time (s)	Acc/ Corr	Time (s)	spd	Acc/ Corr	Time (s)	spd	Acc/ Corr	Time (s)	spd	Acc/ Corr	Time (s)	Sparsity
MRPC [11]	87.01	132	76.47	66	2x	69.36	43	3.07x	86.27	112	1.18x	85.04	133	50%
SST-2 [53]	92.54	2476	91.5	1307	1.89x	86.01	907	2.73x	91.6	1665	1.49x	91.7	2517	60%
CoLA [59]	56.65	309	51.53	150	2.06x	32.98	99	3.12x	56.05	176	1.74x	52.75	318	50%

Table 1: Performance achieved by full fine-tuning, AutoFreeze, static freezing, and Lottery Ticket Hypothesis on MRPC, SST-2, and CoLA datasets. For CoLA dataset, we report the Matthew's Correlation metric. For MRPC and SST-2 datasets, we report the accuracy.

Dataset	Num Train	Num Test	Type
Yelp F. [69]	650,000	50,000	Sentiment
Sogou News [55]	54,000	6,000	Topic
AG's News [69]	120,000	7,600	Topic
IMDb [32]	25,000	25,000	Sentiment
SQuAD2.0 [46]	131,944	12,232	Question
SWAG [67]	73,546	20,006	Multiple Choice
CNN [16]	90,266	1,093	Text Summary
DailyMail [16]	196,961	12,148	Text Summary

Table 2: Statistics and types of datasets used.

intermediate outputs of the BERT encoder to disk in parallel to training when necessary. We implement AutoFreeze in Python and design it to work with PyTorch models [42]. In the distributed fine-tuning setting, the Freezing Module is called on the synchronized gradients on each worker, producing the same freezing decision. When Caching is enabled for the distributed training setting, each GPU manages its own cache to avoid any data movement across machines.

4 Evaluation

We next evaluate AutoFreeze on a number of NLP datasets and tasks, and measure the performance benefits and model accuracy. We compare AutoFreeze to existing baselines and also study scalability by using up to 64 GPUs.

4.1 Datasets, hyper-parameters

In our experimental study we evaluate AutoFreeze on- (i) four text classification datasets, (ii) one question answering dataset, (iii) one multiple choice dataset, (iv) one combined text summarization dataset. The details of the dataset can be found in Table 2. We also use three datasets from the GLUE benchmark, which are all classification tasks, to compare AutoFreeze against Lottery Ticket Hypothesis [6] the details of datasets can be found in Table 1.

Across all the above tasks from (i) to (iii), we set the per epoch evaluation intervals during fine-tuning to 5, and we perform the gradient norm test every $\frac{k}{5}$ iterations (where k = total number of iterations per epoch). As in prior work [18], we used stepped learning rate schedule that decays to slanted triangular learning rate at 0.3 and 0.6 proportions of total iterations from initial learning rate of 1e-5. We set the percentile value for our adaptive freezing algorithm to be the 50th percentile by default unless specified. We run AutoFreeze for three runs with different random seeds for each dataset. In general, when transformer blocks of the BERT Encoder are frozen, we also freeze the Embedding layer as Autograd does not allow backward gradient flow [41] when earlier layers are frozen, i.e., when earlier blocks of the BERT encoder are frozen, the gradients for them are not available, so calculating the gradients for the Embedding layer before the Encoder cannot be achieved. All single GPU experiments are performed on a Azure P100 VM unless otherwise specified.
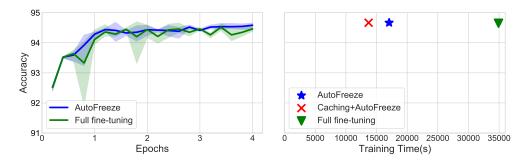
Figure 10: [**AG** (4 epochs)] **Left**: Accuracy achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with caching, and full fine-tuning across three runs. AutoFreeze has $2.05\times$ average improvement in training time when only using the freezing module, and $2.55\times$ when also using caching.

4.2 Comparison with baselines

We compare AutoFreeze with two baseline - (i)Lottery Ticket Hypothesis [6] (LTH) and (ii) static freezing (Table 1).

Lottery Ticket Hypothesis: For comparison we used the same datasets as in prior work [5]. For LTH, we report the fine-tuning time after finding the winning subnetworks by running Iterative Magnitude Pruning. Although it provides comparable accuracy with full fine-tuning, it does not provide training speedup because it utilizes unstructured magnitude pruning where weights are pruned individually instead of a group. Thus, while the winning subnetwork found by the Lottery Ticket Hypotheses is sparse (i.e. has some fraction of weights as 0), it does not provide actual speedup because there is no computation saving due to lack of support for sparse operations in modern accelerators like GPUs and TPUs [64]. Overall we find that AutoFreeze can improve performance by up to $1.74\times$ (Table 1) with minimal degradation in accuracy when compared to existing baselines.

Static Freezing: As shown in the Table 1, static freezing provides significant training speedup but leads to poor accuracy on several different downstream tasks. For example, we observe that freezing up to the 9th layer of the BERT encoder results in around 11% accuracy loss for MRPC, while it only results in around 1% accuracy loss for the SST-2 dataset.

4.3 Freezing, Caching Benefits

To further verify the effectiveness of our freezing scheme in terms of training speedup and model accuracy, we apply AutoFreeze on a number of NLP tasks, and compare it with full fine-tuning of $BERT_{BASE}$. We use a single Azure NC6 VM for these experiments.

Accuracy/F1: In left side of Figures 10, 11, 12, 13, 14, and 15 we plot the mean and the range (max,min) of accuracy values obtained by AutoFreeze as compared to the baseline. We see that the ranges for AutoFreeze overlap with the full fine-tuning line indicating that AutoFreeze is able to achieve comparable accuracy/F1. Similar to prior work [44], we also list the best accuracy/F1 reached across trials for the baseline and AutoFreeze on the right side of Figures 11, 10, 12, 13, 15, and 14. We include complete numbers in the Appendix in Table 5.

From the figures we see that for the Sogou and IMDb datasets, we observe 0.07% and 0.1% reduction in mean of the best accuracy, while for AG News and Yelp F. datasets, we do not see any accuracy loss. From Figure 15, we see an loss of 0.11 in average F1 score for SQuAD v2.0 across three runs. As for SWAG, we observe an accuracy loss of 0.01% in average accuracy.

Training Speedup: As shown in Figure 10, Figure 11, Figure 12, Figure 13, Figure 14, and Figure 15, across all tasks for three independent runs we are able to achieve an average speedup between $1.55\times$-$2.05\times$ with respect to full fine-tuning of $BERT_{BASE}$. We observe AutoFreeze is particularly helpful on large datasets, on AG's News dataset, a large dataset with 120K samples, AutoFreeze is able to save around 5 hours fine-tuning time. For Yelp F. an even larger dataset with 650K data points, we are able to significantly
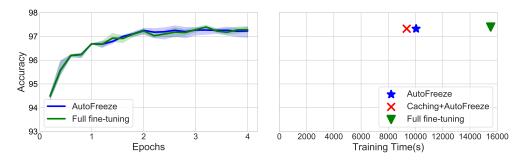
Figure 11: [**Sogou** (4 epochs)] **Left**: Accuracy achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with Caching, and full fine-tuning across three runs. AutoFreeze has $1.55\times$ improvement in training time when only using the freezing module, and $1.66\times$ when also using caching.
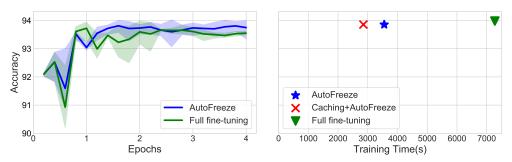

Figure 12: [**IMDb** (4 epochs)] **Left**: Accuracy achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with Caching enabled, and full fine-tuning across three runs. AutoFreeze has $2.05\times$ improvement in training time when only using the freezing module, and $2.55\times$ when also using caching.

reduce the fine-tuning time by around 25 hours.

Caching Benefits Next, we evaluate the speedup gains achieved by switching on both the freezing and caching modules. We see an average speedup of $2.08\times$ across all evaluated tasks compared to the full fine-tuning, which is a $1.14\times$ improvement with respect to average speedup when compared to only using the Freezing module. By enabling caching, we are able to achieve upto $1.25\times$ additional speedup compared to freezing. Generally, we obtain more speedup starting from the third epoch as we start to save the forward pass computation for the frozen layers. For fine-tuning workloads that run for more epochs, the benefits of caching will be more pronounced as shown in our technical report [29].

Caching vs Computation trade-off As described in Section 3.2, when L layers are frozen, the training process consumes data at a rate that corresponds to performing forward and backward computation after layer L while the input reading process operates in parallel to fetch data for the next batch from the cache. Additionally, training process also needs to move data that needs to be written out from GPU to CPU and our measurements show that this adds at most 7% runtime overhead. Thus the balance between caching vs. redoing computation depends on the dataset size, number of layers frozen and computation speed, our storage manager automatically keeps track of these and chooses the best setup. For *e.g.*, as shown in Figure 16a, the overhead from caching fails to be balanced out by the computational savings of skipping part of the forward pass if only the first layer of BERT encoder is frozen. As a result, if the freezing module decides to freeze only the first layer, the storage manager does not activate the caching module.

4.4   Distributed Fine-tuning

In Section 4.3 we show that AutoFreeze achieves significant speedups with minuscule degradation in model accuracy on a single GPU. We next evaluate the benefits of AutoFreeze when performing distributed
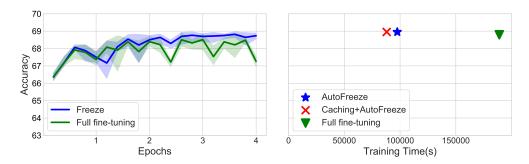
13

Figure 13: [**Yelp F.** (4 epochs)] **Left**: Accuracy achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with Caching enabled, and full fine-tuning across three runs. AutoFreeze has 1.94× improvement training time when only using the freezing module and 2.15× when also using caching.
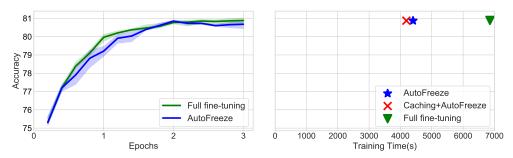

Figure 14: [**SWAG** (3 epochs)] **Left**: Accuracy achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with caching, and full fine-tuning across three runs. AutoFreeze has 1.56× average improvement in training time when only using the freezing module, and 1.64× when also using caching.
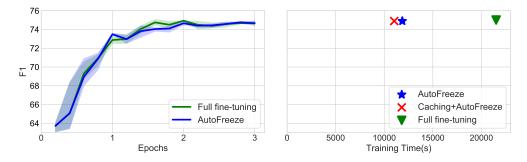

Figure 15: [**SQuADv2.0** (3 epochs)] **Left**: F1 achieved by AutoFreeze and fine-tuning for four epochs. Mean and range of accuracy are plotted. **Right**: Average end-to-end training time for AutoFreeze, AutoFreeze with caching, and full fine-tuning across three runs. AutoFreeze has 1.81× average improvement in training time when only using the freezing module, and 1.95× when also using caching.

fine-tuning.

Setup: For running all our distributed experiments, we used *p3.2xlarge* instances on AWS. We evaluate AutoFreeze in both Performance Packing and Efficiency Packing modes (Section 3.3) on up to 64 GPUs. We observed that the network bandwidth on AWS suffers from random bursts. To minimize the effects of these bursts we used Wondershaper [1] and set the bandwidth to 2Gbps (close to the steady bandwidth).
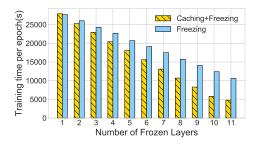
Results: In Figure 17a, we show that Performance Packing is able to reduce the end-to-end training time by 4.38× for AG's News and 4.74× for Sogou News datasets when running on 64 GPUs. Efficiency Packing can reduce the end-to-end training time by 3.55× for AG's News and 3.44× for Sogou News when compared
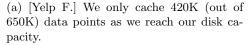
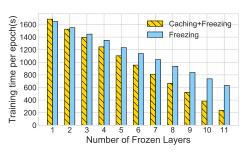|	Full fine-tuning	Eff packing	Perf packing
AG's News	94.43	94.54	94.42
Sogou News	97.43	97.23	96.9

Table 3: [64 GPUs] Accuracy for full fine-tuning, AutoFreeze with Efficiency Packing, and AutoFreeze with Performance Packing.

to performing full fine tuning. As for the total cost, Efficiency Packing reduces the total cost by $5.03\times$ and $5.21\times$ for AG's News and Sogou News datasets, while Performance Packing reduces the total cost by $4.38\times$ and $4.74\times$ for AG's News and Sogou News datasets when compared against full fine tuning.

Breaking down the benefits, when we freeze 11 BERT layers for an epoch with the AG's News dataset, the average iteration time for Performance Packing with 64 machines is 1.05 seconds and there are 35 mini-batches in an epoch. On the other hand, the average iteration time for Efficiency Packing on 8 machines is 0.42 seconds but runs 313 mini-batches in an epoch.

(a) [Yelp F.] We only cache 420K (out of 650K) data points as we reach our disk capacity.

(b) [IMDb] We cache the whole dataset, containing 25K points, to CPU memory.

Figure 16: Trade-off between caching and running the full forward pass as we vary the number of frozen layers.
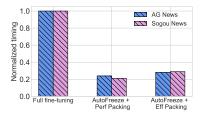
We also measured the accuracy for both Efficiency Packing and Performance Packing and compare them to full fine-tuning without changing number of GPUs or per GPU batch size on AG's News and Sogou datasets (Table 3). AutoFreeze with Efficiency Packing results in negligible or no accuracy loss compared to full fine-tuning, while AutoFreeze with Performance Packing can result in at most 0.5% accuracy loss. Performance Packing is more prone to accuracy loss as adaptively increasing batch size on freezing of layers can lead to extremely large batches which has been shown to cause accuracy drops [65]. As mentioned in Section 3.3, users of AutoFreeze can configure a maximum batch size to avoid accuracy drops.
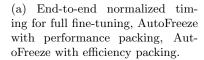
Finally, we also study how the benefits of AutoFreeze changes as we scale from 8 to 64 GPUs. As shown in Figure 17c, we can see that increasing the number of GPUs used in fine-tuning can lead to almost linear decrease in end-to-end fine-tuning time. We also observe that AutoFreeze can achieve similar speedup (varying from $4.10\times$-$4.31\times$) across different number of GPUs when compared to full fine-tuning.
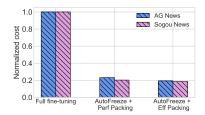
4.5 Ablation Studies

We next study how the benefits from AutoFreeze change as we consider more complex training tasks and newer hardware. We also study the sensitivity of AutoFreeze to various parameters and the overheads from the gradient norm test.

AutoFreeze on Text summarization: We test how well AutoFreeze works on more complex tasks by considering a text summarization problem. We follow the experimental setup and hyper-parameters in [28] and train the BERTSUMABS model for 200000 steps on four NVIDIA V100 GPUs. We set the total number of evaluation intervals to 20 and evaluate every 10000 steps during fine-tuning. In Table 4 we show the Recall-Oriented Understudy for Gisting Evaluation (ROUGE) F score, which is a measurement for the similarity between a candidate document and reference documents. As shown in Table 4 we observe negligible loss

(a) End-to-end normalized timing for full fine-tuning, AutoFreeze with performance packing, AutoFreeze with efficiency packing.

(b) End-to-end normalized total cost for full fine-tuning, AutoFreeze with performance packing, AutoFreeze with efficiency packing.

(c) [AG's News] End-to-end training time for a fixed freezing pattern over fine-tuning for 4 epochs using 8, 16, 32, and 64 GPUs.

Figure 17

Figure 18: [**Vision Models**] Performance of full fine tuning and AutoFreeze when fine tuning ResNet-18 trained on CINIC-10 with CIFAR-10 dataset. AutoFreeze achieves similar accuracy while being 2.15× faster.
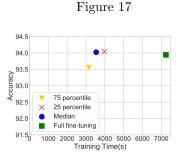
Figure 19: [**IMDb**] Comparison between freezing schemes when using 75th percentile, 25th percentile and median in Algorithm 1.
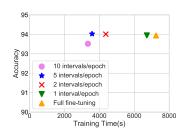
Figure 20: [**IMDb**] Comparison between freezing schemes by varying number of evaluation intervals for each epoch from 1 interval/epoch to 10 intervals/epoch.

(less than 0.15) in ROGUE F1 score when comparing AutoFreeze and full fine-tuning. Further, AutoFreeze is able to achieve a speedup of 1.55× comparing to full fine-tuning.

CINIC-10 transfer learning: Although BERT fine tuning is our primary focus, we also evaluate AutoFreeze on the vision task described in Section 2. We take a ResNet-18 model which is pre-trained with CINIC-10 and fine-tune this model for the CIFAR-10 dataset. In Figure 18 we observe that our freezing scheme is able to reduce the fine-tuning time by more than 2× and still reaches a very similar accuracy as of full fine tuning. This demonstrates that our freezing scheme is applicable on tasks other than BERT and we plan to explore these domains in detail the future.

Using A100 GPUs: To show AutoFreeze can provide the benefits even on advanced hardware, we tested AutoFreeze on the latest NVIDIA A100 GPU. We set up the experiments on a p4d.24xlarge instance with eight A100s on AWS, and ran AutoFreeze on the AG's News dataset. To match the total batch size of 24 [55] for classification tasks, we set the per GPU batch size to 3. While full fine-tuning took 3030 seconds for four epochs, AutoFreeze only took 1670 seconds and reached similar accuracy. Thus, we see a speed-up of 1.81× on A100s which is comparable with the speedup achieved on P100s and shows that the benefits of AutoFreeze remain across hardware generations.

Finally, we also ran the experiment to fully utilize the memory of the A100 GPUs by setting the per GPU batch size to 16, and fine-tune $BERT_{LARGE}$ on AG News. AutoFreeze is able to finish fine-tuning in 1304 seconds compared to 2020 seconds for full fine-tuning (1.54× speed-up).

Using different percentiles: As shown in Figure 19, we observe a trade-off between training speedup gains and model accuracy as we vary the percentile threshold used in Algorithm 1. We observe, the more aggressive our freezing policy, the greater the accuracy loss will be. For example, if we set N to be 75th percentile which results in a more aggressive policy, we get 0.39% accuracy loss for the IMDb dataset. However, we do not see accuracy loss when we set N to be 25th or 50th percentile. On the other hand,

16

	Text Summarization			
	R1	R2	RL	Training Time (s)
AutoFreeze	41.31	18.9	38.46	116528
Full fine-tuning	41.45	19.02	38.55	180450

Table 4: [**Text Summarization**] ROUGE F score results on CNN/DailyMail test set (R1 and R2 stand for unigram and bigram overlap; RL, the longest common subsequence). We took top-3 checkpoints based on the performance on the validation set and report the average results on the test set.

increasing N gives us more training speedup. A policy with $N = 75$ achieves 20% more speedup compared with the $N = 25$ policy with 0.49% difference in max accuracies achieved. For all experiments in this paper we chose 50th percentile and have not fine tuned this parameter.

Varying number of evaluation intervals each epoch: We also vary how frequently the freezing module is invoked and Figure 20 shows the results with the IMDb dataset. We see that if the frequency is too low (e.g., 1 interval/epoch) then the speedup obtained is limited. On the other hand, using 10 intervals/epoch results in gradient vectors that are not fully representative, thus leading to a drop in accuracy. However, we find this trade-off is balanced for a range of values (2 to 5 intervals/epoch).

Gradient Norm Test Overheads: As stated in Section 3.1.1, the gradient norm test has minimal overhead. The overhead mainly comes from accumulating the gradient vectors within an evaluation interval T. For example, for IMDb, the overhead for gradient accumulation in terms of time is 13 seconds for every interval, which is less than 1% of the execution time of an interval. The memory overhead for storing the gradients is 453MB for an epoch.

5 Related Work

Previous works have largely focused on reducing the size of pre-trained models or improving the accuracy/stability for fine-tuning.

Improving Fine-Tuning Several methods [36, 38, 14] have been developed in NLP literature to achieve good accuracy when fine tuning. [18] introduce ULFiT, a technique which has enabled state of the art performance when doing fine tuning. Similarly [55] investigates fine-tuning methods of BERT on text classification tasks including layer selection, layerwise learning rate, and multi-task Learning. [55] shows lower layers contain more general information, and using features from the last layer of BERT gives the best fine-tuning accuracy. [20] proposes a smoothness inducing regularizer for improving robustness during training. Similarly other works have proposed different techniques [43, 17, 54] aiming to improve accuracy while reducing parameters of the final trained model. In this work, our aim is to speed up time for fine tuning without losing the accuracy gains provided by techniques proposed in [18]. We show that AutoFreeze achieves the same accuracy as these methods but reduces time for fine-tuning by adaptively freezing layers at run time.

Model Compression The primary goal of several previous works is to reduce the size of the fine tuned model to enable fast inference. [23, 31] perform low rank approximations to reduce the size and compute requirements of the model. [49, 22, 56] use knowledge distillation to train a smaller model. [62] uses multiple teacher models to train a student for multi-task learning. Similarly [44, 4, 66] perform quantization to enable fast compression. On the other hand [8, 35, 37] reduce the model size using pruning. Similarly [12] use structured dropout to introduce sparsity. The goal of these methods is to reduce the time for inference which sometimes lead to more time spent in training. On the other hand in this work we aim to to reduce time for fine tuning of the BERT model for new tasks.

Adaptive ML: Another line of work shows that certain layers of network can be skipped dynamically during inference to reduce inference time. [61, 27, 51] propose techniques for adaptively skipping layers at inference. Another recent work [2] uses gradient norms to adaptively tune communication in distributed learning. On

other hand our work is focused on speeding up fine-tuning with adaptive freezing.

Speeding up Pre-training Gong et al. [13] present a method to speed up BERT pre-training by progressively increasing the size of the model by stacking layers. Zhang and He [68] propose speeding up of BERT pre-training by progressively dropping the layers during pre-training. Chen et al. [7] introduces Early-BERT [7] which extends the work done on finding lottery-tickets in CNNs [64] to speedup both pre-training and fine-tuning for BERT models. Experimental evaluation of EarlyBERT [7] shows some degradation in accuracy for fine-tuning unlike our work where we have almost no accuracy loss, while providing almost similar speedups.

6   Conclusion and Future Work

Fine-tuning pre-trained models has become a popular and accurate method for developing ML models for new tasks. However there are a number of performance challenges in fine-tuning. In this paper, we proposed AutoFreeze, a scheme to adapatively freeze parts of the model that are closest to convergence during fine-tuning. We show that using AutoFreeze on NLP tasks can give up to 2.55x speed up on a single GPU and 4.38× in a 64 GPU cluster without affecting accuracy. While this paper mainly focused on BERT due to its popularity, we plan to study if similar approaches also help in other domains like image classification or speech recognition. Our implementation is available at `https://github.com/uw-mad-dash/AutoFreeze`.

References

[1] The wonder shaper 1.4.1. `https://github.com/magnific0/wondershaper`. Accessed: February 22, 2020.

[2] S. Agarwal, H. Wang, K. Lee, S. Venkataraman, and D. Papailiopoulos. Accordion: Adaptive gradient communication via critical learning regime identification. *arXiv preprint arXiv:2010.16248*, 2020.

[3] A. Bansal, K. Sikka, G. Sharma, R. Chellappa, and A. Divakaran. Zero-shot object detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 384–400, 2018.

[4] A. Bie, B. Venkitesh, J. Monteiro, M. Haidar, M. Rezagholizadeh, et al. Fully quantizing a simplified transformer for end-to-end speech recognition. *arXiv preprint arXiv:1911.03604*, 2019.

[5] H. Chen, Y. Wang, G. Wang, and Y. Qiao. Lstd: A low-shot transfer detector for object detection. *arXiv preprint arXiv:1803.01529*, 2018.

[6] T. Chen, J. Frankle, S. Chang, S. Liu, Y. Zhang, Z. Wang, and M. Carbin. The lottery ticket hypothesis for pre-trained bert networks. *arXiv preprint arXiv:2007.12223*, 2020.

[7] X. Chen, Y. Cheng, S. Wang, Z. Gan, Z. Wang, and J. Liu. Earlybert: Efficient bert training via early-bird lottery tickets. *arXiv preprint arXiv:2101.00063*, 2020.

[8] B. Cui, Y. Li, M. Chen, and Z. Zhang. Fine-tune bert with sparse self-attention mechanism. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3539–3544, 2019.

[9] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey. Cinic-10 is not imagenet or cifar-10. *arXiv preprint arXiv:1810.03505*, 2018.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] W. B. Dolan and C. Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005. URL `https://www.aclweb.org/anthology/I05-5002`.

[12] A. Fan, E. Grave, and A. Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[13] L. Gong, D. He, Z. Li, T. Qin, L. Wang, and T. Liu. Efficient training of bert by progressively stacking. In *International Conference on Machine Learning*, pages 2337–2346, 2019.

[14] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964*, 2020.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] K. M. Hermann, T. Kočiský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom. Teaching machines to read and comprehend, 2015.

[17] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly. Parameter-efficient transfer learning for nlp. *arXiv preprint arXiv:1902.00751*, 2019.

[18] J. Howard and S. Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1031.

[19] L. Huang, H. Ji, K. Cho, and C. R. Voss. Zero-shot transfer learning for event extraction. *arXiv preprint arXiv:1707.01066*, 2017.

[20] H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and T. Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437*, 2019.

[21] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/jiang.

[22] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.

[23] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[24] J. Lee, R. Tang, and J. Lin. What would elsa do? freezing layers during transformer fine-tuning. *arXiv preprint arXiv:1911.03090*, 2019.

[25] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[26] S. Liang, Y. Li, and R. Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks, 2020.

[27] W. Liu, P. Zhou, Z. Zhao, Z. Wang, H. Deng, and Q. Ju. Fastbert: a self-distilling bert with adaptive inference time. *arXiv preprint arXiv:2004.02178*, 2020.

[28] Y. Liu and M. Lapata. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*, 2019.

[29] Y. Liu, S. Agarwal, and S. Venkataraman. Autofreeze: Automatically freezing model blocks to accelerate fine-tuning. *arXiv preprint arXiv:2102.01386*, 2021.

[30] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJlnB3C5Ym.

[31] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song. A tensorized transformer for language modeling. In *Advances in Neural Information Processing Systems*, pages 2232–2242, 2019.

[32] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[33] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

[34] D. Madras, J. Atwood, and A. D'Amour. Detecting extrapolation with local ensembles. *arXiv preprint arXiv:1910.09573*, 2019.

[35] J. S. McCarley. Pruning a bert-based question answering model. *arXiv preprint arXiv:1910.06360*, 2019.

[36] N. Miao, Y. Song, H. Zhou, and L. Li. Do you have the right scissors? tailoring pre-trained language models via monte-carlo methods. *arXiv preprint arXiv:2007.06162*, 2020.

[37] P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one? In *Advances in Neural Information Processing Systems*, pages 14014–14024, 2019.

[38] S. Murty, P. W. Koh, and P. Liang. Expbert: Representation engineering with natural language explanations. *arXiv preprint arXiv:2005.01932*, 2020.

[39] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A data system for optimized deep learning model selection. *Proc. VLDB Endow.*, 13(12):2159–2173, July 2020. ISSN 2150-8097. doi: 10.14778/3407790. 3407816. URL https://doi.org/10.14778/3407790.3407816.

[40] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

[41] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[43] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[44] C. Qiao, B. Huang, G. Niu, D. Li, D. Dong, W. He, D. Yu, and H. Wu. A new method of region embedding for text classification. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BkSDMA36Z.

[45] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep understanding and improvement. *ArXiv*, abs/1706.05806, 2017.

[46] P. Rajpurkar, R. Jia, and P. Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

[47] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[48] Y. Roh, G. Heo, and S. E. Whang. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[49] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[50] S. Sarvotham, R. Riedi, and R. Baraniuk. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, page 99–103, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134355. doi: 10.1145/ 505202.505215. URL https://doi.org/10.1145/505202.505215.

[51] R. Schwartz, G. Stanovsky, S. Swayamdipta, J. Dodge, and N. A. Smith. The right tool for the job: Matching model and instance complexities. *arXiv preprint arXiv:2004.07453*, 2020.

[52] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[53] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, Oct. 2013. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/D13-1170`.

[54] A. C. Stickland and I. Murray. Bert and pals: Projected attention layers for efficient adaptation in multi-task learning. *arXiv preprint arXiv:1902.02671*, 2019.

[55] C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune BERT for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer, 2019.

[56] S. Sun, Y. Cheng, Z. Gan, and J. Liu. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*, 2019.

[57] A. Suprem, J. Arulraj, C. Pu, and J. Ferreira. Odin: automated drift detection and recovery in video analytics. *arXiv preprint arXiv:2009.05440*, 2020.

[58] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[59] A. Warstadt, A. Singh, and S. R. Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[60] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, Oct. 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

[61] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

[62] Z. Yang, L. Shou, M. Gong, W. Lin, and D. Jiang. Model compression with multi-task knowledge distillation for web-scale question answering system. *arXiv preprint arXiv:1904.09636*, 2019.

[63] W. Ying, Y. Zhang, J. Huang, and Q. Yang. Transfer learning via learning to transfer. In *International Conference on Machine Learning*, pages 5085–5094, 2018.

[64] H. You, C. Li, P. Xu, Y. Fu, Y. Wang, X. Chen, R. G. Baraniuk, Z. Wang, and Y. Lin. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJxsrgStvr`.

[65] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, and C.-J. Hsieh. Large-batch training for lstm and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2019.

[66] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat. Q8bert: Quantized 8bit bert. *arXiv preprint arXiv:1910.06188*, 2019.

[67] R. Zellers, Y. Bisk, R. Schwartz, and Y. Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. *arXiv preprint arXiv:1808.05326*, 2018.

[68] M. Zhang and Y. He. Accelerating training of transformer-based language models with progressive layer dropping. *arXiv preprint arXiv:2010.13369*, 2020.

[69] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems*, pages 649–657, 2015.

[70] J. Zhu, Y. Xia, L. Wu, D. He, T. Qin, W. Zhou, H. Li, and T.-Y. Liu. Incorporating bert into neural machine translation. *arXiv preprint arXiv:2002.06823*, 2020.

[71] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[72] E. Zisselman and A. Tamar. Deep residual flow for out of distribution detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13994–14003, 2020.

A   Appendix

A.1   Complete Results

Table 5 shows the max accuracy, iterations at which the max accuracy is achieved, and the end-to-end fine-tuning time for AutoFreeze and full fine-tuning across three random trials for the classification tasks. Table 6 shows the max F1, iterations at which the max F1 score is reached, and the end-to-end fine-tuning time for AutoFreeze and full fine-tuning for SQuAD2.0 dataset. Table 7 shows similar measurements for the SWAG dataset. We observe that we gain similar speedup and max accuracy for all the runs for each dataset.

We also include the test accuracy convergence curve with respect to time for each of the three repeated runs using stepped learning rate schedule for each dataset in Figures 23, 24, 25, 26, 27, and 28. We see that AutoFreeze and full fine-tuning achieve comparable max accuracy with an average end-to-end training speedup of 2.05×, 1.55×, 2.05×, 1.94×, 1.81×, and 1.56× for AG News, Sogou News, IMDb, Yelp F., SQuAD2.0 and SWAG respectively. We can also see that the freezing speedup is on the same scale across different runs. We gain 2.55×, 1.66×, 2.55×, 2.15×, 1.95×, and 1.64× more speedup on average when turning on the storage manager for AG News, Sogou News, IMDb, and Yelp F., SQuAD2.0 and SWAG respectively compared to full fine-tuning. As shown in Figure 24(b), we do not have significant improvements when turning on the storage manager as AutoFreeze decides to start freezing layers from the third epoch for this set of experiments. Accordingly, we are only able to achieve speedup gains from the last epoch through caching.
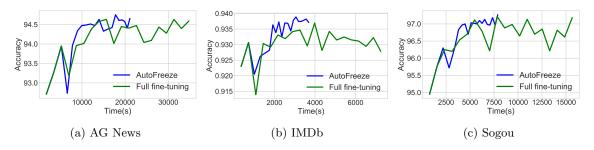

Figure 21: Test accuracy curve with respect to end-to-end training time for AutoFreeze and full fine-tuning when using **constant learning rate** schedule.

A.2   Constant learning rate schedule

To show that AutoFreeze is effective for other learning rate schedules, we run AutoFreeze using constant learning rate schedule with learning rate of 1e-5. As shown in Figure 21a, 21b and 21c, AutoFreeze is able to achieve 1.65×, 1.96×, and 1.98× speedup for AG News, IMDb, and Sogou News with respect to end-to-end fine-tuning time without harming the model accuracy.

A.3   Caching benefits for longer runs

We next present the benefits brought by caching when we fine-tune $BERT_{BASE}$ for more than four epochs. We fine-tune $BERT_{BASE}$ on Sogou dataset for 6 epochs using stepped learning rate schedule with initial learning rate of 1e-5. As shown in Figure 22, AutoFreeze is able to achieve 2.16× speedup compared to the full fine-tuning. When the storage manager is turned on, we get 3.01× speedup compared with the baseline. The storage manager is able to get more significant speedup in this setup as AutoFreeze decides to freeze up to layer 9 at the end of the second epoch, thus saving most of the forward computation for future epochs. In general, we can save more forward computation time when we run the fine-tuning procedure for longer epochs. However, for the datasets we consider in this paper we use a maximum of four epochs. This is because, as reported in prior work [55], using more epochs doesn't lead to significant improvements in accuracy for these datasets.
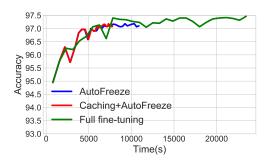
Figure 22: Benefits from AutoFreeze when the **Sogou** dataset is fine-tuned for more epochs (6 epochs). We see that caching can provide more benefits in this case.

Dataset	AutoFreeze			Full fine-tuning			Training Speedup
	Best Iteration	Accuracy	Training Time(s)	Best Iteration	Accuracy	Training Time(s)	
AG News	80000	94.66	18993	40000	94.59	34559	1.82×
	28000	94.68	15936	52000	94.66	35114	2.20×
	80000	94.66	16242	36000	94.70	35058	2.16×
Sogou News	21600	97.45	10795	28800	97.38	15552	1.44×
	30600	97.12	9462	28800	97.32	15527	1.64×
	28800	97.4	9866	28800	97.48	15478	1.57×
Yelp F.	389988	68.96	97368	324990	68.83	188892	1.94×
	389988	68.63	102859	194994	68.44	189207	1.84×
	303324	68.94	92226	281658	68.91	188957	2.05×
IMDb	9163	93.94	3543	4165	93.944	7304	2.06×
	10829	94.024	3584	4165	93.944	7267	2.03×
	15827	93.604	3512	8330	93.98	7253	2.07×

Table 5: **AutoFreeze Performance Evaluation (Classification tasks):** We report performance of AutoFreeze on 4 different datasets. Each experiment is repeated 3 times with different random seeds. We observes AutoFreeze leads to upto 2× reduction in fine tuning time while reaching same accuracy as full fine tuning.

Figure 23: [**AG News**] Test accuracy curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.

Dataset	AutoFreeze			Full fine-tuning			Training Speedup
	Best Iteration	Dev F1	Training Time (s)	Best Iteration	Dev F1	Training Time (s)	
SQUAD2.0	42881	74.83	11002	29687	74.90	21419	1.94x
	29687	75.02	12163	29687	74.95	21532	1.77x
	42881	74.78	12414	23090	75.05	21512	1.73x

Table 6: **AutoFreeze Performance Evaluation (Question Answering):** We report performance of AutoFreeze on SQuAD2.0. The experiment is repeated 3 times with different random seeds.

Dataset	AutoFreeze			Full fine-tuning			Training
	Best Iteration	Dev F1	Training Time (s)	Best Iteration	Dev F1	Training Time (s)	Speedup
	4138	80.85	4436	6437	80.72	6868	1.55x
SWAG	6437	81.02	4663	6437	80.89	6848	1.47x
	4138	80.88	4107	5057	80.92	6857	1.66x

Table 7: **AutoFreeze Performance Evaluation (Multiple Choice):** We report performance of AutoFreeze on the SWAG dataset. Each experiment is repeated 3 times with different random seeds.

Figure 24: [**Sogou News**] Test accuracy curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.

Figure 25: [**IMDb**] Test accuracy curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.
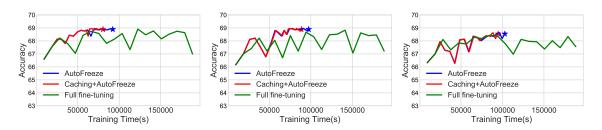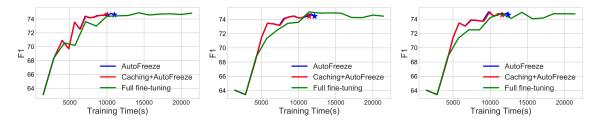

Figure 26: [**Yelp F.**] Test accuracy curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.

Figure 27: [**SQUAD2.0**] Dev F1 curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.

Figure 28: [**SWAG**] Dev accuracy curve for each trial with respect to end-to-end training time for AutoFreeze, AutoFreeze with Caching turned on, and full fine-tuning.