Fast & Efficient DNN Inference Using Practical Early-Exit Networks

Abstract

Inference using Deep Neural Networks (DNN) has emerged as the de-facto standard for many applications today. To meet the stringent latency requirements, existing approach to performing inference is to use model compression techniques, such as pruning, quantization and distillation. Early-exit networks, an alternative, orthogonal approach, has gained traction in the machine learning community, but face fundamental challenges that make them hard to deploy. In this paper, we present TETRIS, a system that makes early-exit networks practical and uses them to enable fast and efficient inference. TETRIS incorporates an online batch profile estimator that identifies the batching characteristics. TETRIS then proposes splitting the model into smaller pieces and executing them in a model-parallel, pipelined fashion on heterogeneous resources ensuring that the combination of splits maintain a constant batch size by posing the splitting and placement as an optimization problem. We evaluate TETRIS against state-of-the-art early-exit and stock models in NLP and vision, and our experiment results show that it is able to obtain up to 74% improvement in goodput and up to 78% reduction in cost.

1 Introduction

As modern user-focused applications increasingly depend on Machine Learning (ML) to improve their efficacy, ML inference, the process of deploying trained models and serving live queries using them, has become the dominant and critical workload in many real-world applications [11]. Industry scale ML inference systems currently serve billions of queries per day, which translates to many 1000s of queries *per second*, and require the use of massive clusters of powerful GPUs [8, 24]. As a result, ML inference pipelines incur significant cost [19].

The high cost of ML inference is exacerbated by the fact that the requirements for inference differs drastically from that of training. While ML training is throughput intensive, inference is both throughput and *latency* sensitive. Since inference systems are user-facing, they operate under stringent Service Level Objectives (SLOs) that dictate the maximum latency allowed for each query—typically under 100 milliseconds—so as to not hinder user-experience. Such stringent budgets combined with the increase in model sizes, as they continue to improve, translate to even more costly resources in the inference infrastructure. Thus, significant efforts have been made to reduce the resource requirements for ML inference [19].

Existing approaches to reducing the inference latency, such as distillation [12, 15], pruning [6, 7] and quantization [25] have focused on *compressing or reducing the size of the model*. Since the execution time of the model is directly proportional to its size, a smaller model can thus be deployed on a smaller, less powerful resource. Distillation is based on the idea that larger models have vast knowledge that may not be fully utilized for a given workload [12, 15]. Consequently, the larger, complex model is replaced with a cheaper, significantly smaller model by *transferring knoweldge* from the larger model. As an example, DistilBERT [22], a distilled version of the popular language model BERT [5] is ≈40% smaller and 60% faster. Distillation is often used in conjunction with pruning and quantization—removal of weights and use of low-precision arithmetic, respectively—to achieve even more compression. However, compression techniques face three shortcomings. First, they incur some accuracy loss due to the removal of layers and/or parameters. Since the amount of loss is determined by the amount of compression, these techniques pick a fixed point in the accuracy-latency trade-off curve. Second, since they are often tuned to specific workloads, changes may lead to expensive retraining. Finally, even a compressed model may be an overkill for the workload under consideration.

An alternative approach that has garnered attention in the ML community is *early-exit networks* [18, 23, 28–31] (EE-DNN[1]), which proposes the idea that inputs to a DNN can exit at any point and not traverse through all the layers: easy inputs can exit early while hard inputs continue through the end. This results in the optimal execution time for any given input, as early-exit networks dynamically adapt to the variability in hardness of the workload. While early-exit networks may seem like the perfect candidate for inference, a fundamental limitation restricts their use in practice. The natural solution to improving resource utilization and increasing goodput in ML is to use *batching* for the input. However, since each input in a batch can exit at different points in the EE-DNN, over the course of execution the batch size decreases dramatically. This results in substantial drop in resource utilization leading to poor performance, in many cases worse compared to not using early-exits altogether. Consequently, state-of-the-art early-exit systems have disabled the use of batching [18, 23, 28–31], making them hard to deploy in the real-world. (§2)

In this paper, we propose TETRIS, a system that makes early-exit networks practical and leverages it to enable fast

[1] In the rest of this paper, we use EE-DNN to refer to a DNN model augmented with early-exits. Early exits also apply to compressed models.

and cost-effective inference. The key idea in TETRIS is simple: it *maintains the batch size constant* throughout the execution of the EE-DNN. By maintaining a constant batch size and not allowing it to shrink over the course of execution, TETRIS is able to avoid the fundamental inefficiency associated with EE-DNNs, making them practical for real-world deployments and attaining substantial performance gains.

While the idea in TETRIS is seemingly simple, achieving it is not. To reach its goal, TETRIS incorporates three techniques. First, TETRIS observes that workloads vary over time, and as a result, not all the exits in a EE-DNN are useful at all times. Exploiting this, TETRIS proposes an online batch profile estimation technique, based on ARIMA [13], that can predict how the batch size shrinks over the execution of the EE-DNN model with high confidence. Using the estimated batch profile as a guide, TETRIS then proposes *splitting* the EE-DNN model into smaller pieces and executing them independently at different batch sizes so that combining the pieces result in a constant batch size.(§ 3.1)

Although the splits of a EE-DNN could be run on a single GPU, the ability to run the splits independently enables TETRIS to incorporate a *inter-layer model parallel* scheduler to execute them in a parallel fashion. While model-parallelism is not typically used in ML inference due to their communication overhead, TETRIS embraces it to its advantage. Even with the additional communication incurred due to model parallelism, TETRIS is able to provide significant gains. TETRIS further reduces the overhead of communication by leveraging *pipelining* to overlap computation and communication across batches.(§ 3.3)

To enable efficient model parallelism, TETRIS needs to determine the right number and location of the splits in the EE-DNN model, and the optimal amount of resources that are needed to run them. Additionally, all of TETRIS's decisions must adhere to the strict SLO constraints. Towards this, TETRIS uses its online batch profile estimation as a guideline to build a Dynamic Programming (DP) based optimization formulation. TETRIS's DP formulation considers the potential exits, the execution time of each individual splits among all possible splits, the available resources, and the communication overheads to determine the correct number of splits, the right resources to run them on, and the optimal batch size for each individual split that maximizes goodput while satisfying the SLO and other constraints.(§ 3.2)

Finally, TETRIS observes that by splitting up the EE-DNN, it now has the opportunity to leverage *heterogeneous* hardware to execute them. Based on this, it proposes a simple modification to its DP formulation that incorporates heterogeneity of the resources. We notice that this results in substantial reduction in inference cost for the same throughput compared to not using EENs, or significantly improves throughput for the same cost. To the best of our knowledge, TETRIS is the first system to exploit heterogeneity and early-exit networks to provide substantial benefits for large scale inference.

The combination of these techniques allow TETRIS to significantly outperform existing state-of-the-art EE-DNNs. We have implemented TETRIS in PyTorch and have evaluated it on a number of different early-exit architectures, which capture the spectrum of architectures proposed today. On a wide variety of workloads, TETRIS is able to achieve up to 74% higher goodput at the same cost, or reduce the cost by up to 78% compared to state-of-the-art EE-DNNs. Additionally, TETRIS is able to outperform stock DNN models while existing EE-DNN models fail to do so. (§5)

- We propose TETRIS, a system that supports fast and resource-efficient inference by leveraging early-exit networks, making them practical. To the best of our knowledge, TETRIS is the first system to combine practical early-exit networks and resource heterogeneity to significantly accelerate inference performance.
- TETRIS exposes the challenges in making early-exit networks practical, and proposes novel techniques to resolve them. It incorporates an online batch size profiler based on ARIMA [13], an inter-layer model parallelism based scheduler that can take advantage of resource heterogenity, and a dynamic programming based optimization formulation to determine the right splits, and the resources to run them on that optimizes goodput while adhering to strict SLO constraints.
- We show through extensive evaluation that TETRIS is able to significantly outperform existing state-of-the-art EE-DNN and stock DNN models, and that it is able to utilize the resources optimally.

2 Background

We begin with a brief background on model compression techniques and early-exit networks, then motivate the challenges with making early-exit networks practical.

2.1 Model Compression

The quest towards improvement in accuracy has led ML models to steadily increase in complexity, mainly in the form of deeper architectures (more layers) and large number of parameters. For instance, TuringNLG [3], a popular language model consists of 17 billion parameters. Using such complex models directly for inference is often not possible: even with the most power accelerators available, it may not be possible to meet the SLOs necessary for the user-facing application. Model compression has sought to resolve this problem by proposing techniques to replace the original, complex model with a simpler form without significant reduction in accuracy. The key insight exploited by model compression is the observation that while the original model has significant predictive power, only a fraction of is used for an inference task.

The most commonly used techniques for model compression include pruning, quantization and distillation. Pruning is based on the notion that models are often over-parameterized, and hence *pruning* the unnecessary parameters can result

in a model that is smaller. Pruning can also be applied to neurons and layers in the network in addition to weights. Although beneficial, pruning can often be computationally intensive [6, 7]. Quantization, on the other hand, aims to reduce the size of the model by reducing the amount of storage necessary for the weights. By manipulating how weights are represented, the amount of memory required to hold the weights can be significantly reduced. For instance, replacing weights originally in 32-bit representation by a binarization process [25] can lead to 32× reduction in size. However, quantization can lead to substantial loss of accuracy.

Knowledge distillation has emerged as a popular compression technique in the recent past, where a smaller model is trained using knowledge *distilled* from the original model. Here, the smaller model, referred to as the *student* model is trained to mimic the larger model, referred to as the *teacher* model, using the output from the original model. At a high level, the student model learns the function the teacher has learned from its training, aided by the teacher. Several methods of distillation have been proposed, including collaborative learning and assistant models, each with their own pros and cons. In general, distillation can produce substantially smaller models with very little sacrifice in accuracy. For instance, DistilBERT [22] is a ≈40% smaller model compared to BERT [5] but retains 95% of its performance. However, distillation has two disadvantages. First, it loses some accuracy with respect to the original model. The amount of loss depends on how faithfully the student model mimics the teacher: a strictly accurate mimicking would lead to a larger size of the student model (and hence less compression) also to become large and vice-versa. This means that like all compression schemes, distillation also picks a fixed point in the latency accuracy curve. Second, depending on the fixed point, the distilled model itself may be overly complex for a given inference workload.

2.2 Early-Exit Networks

Early-exit networks (fig. 1) is based on the idea that a model's predictive power is utilized to various degrees by individual inputs. That is, in a given inference workload, the *hardness* of the queries varies: some queries are simple, some hard and some of medium difficulty. A hard query may use the full predictive power of the model, but the easy examples do not. Early-exit networks puts forward the idea that the non-hard inputs can be predicted accurately by the model with *less work*, or in other words, they can exit the model before they reach the normal end-point. Since the latency of executing a model is directly proportional to the number of layers, exiting earlier translates to a lower latency.

An ideal early-exit network would, in theory, incur the optimal amount of latency for any given input; at the same time, it alleviates the shortcomings with other compression techniques since hard queries can still benefit from the predictive power of the original model. However, in practice, a decision to exit early has to be made. Typically, this is done by computing an entropy of the output of a layer, using techniques ranging from simple computation to deploying an entire neural network for the task. Due to this, early-exit networks also incur an accuracy loss. However, compared to aforementioned techniques for compression, early-exit networks allow a smooth traversal of the latency-accuracy curve. Note that early-exit is orthogonal to the compression techniques, as a pruned, quantized and distilled model can also be made to be an early-exit network.

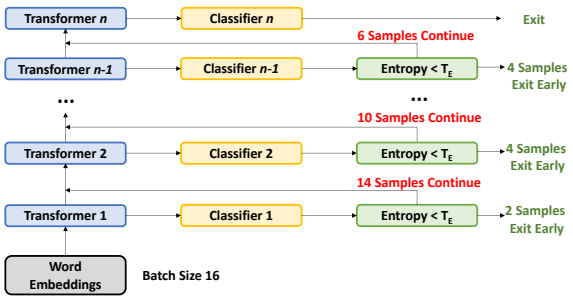


Figure 1: Early-exit DNNs allow inputs to exit "early", thus reducing inference latency. Picture depicts inputs exiting at different layers.

Machine learning researchers have proposed many forms of early-exit networks, where the techniques differ in how they determine to exit at a given layer. The exit point is often referred to as a *ramp*. The simplest ramp is an entropy computation that provides the confidence of the prediction at that point. More complex early-exit architectures include counter-based mechanisms, which count the confidence of the last k layers before deciding to exit, and neural network based ramps which take as input the output from earlier layers.

2.3 Challenges

Although early-exit networks seem compelling with their near-ideal characteristics suited for inference, there are many challenges in making early-exit networks practical.

Challenge #1: Overhead of ramps. While early-exit networks provide the optimal exit point, they need to check if an input can exit at a given layer. This check incurs some overhead in terms of computation time. With a model with large number of layers, the overhead can add up and result in becoming a bottleneck. For example, a hard example that has to pass through all the layers will incur *more latency* compared to not having early exit ramps at all. early-exit networks have proposed adding ramps only at certain layers based on their importance. Unfortunately, determining this is a challenge in itself, and eliminates the advantage of early-exits.

Challenge #2: Batching. A fundamental requirement in achieving optimal throughput, in both ML training and inference, is the ability to batch the input. Batching enables accelerators, such as GPUs, to utilize all the cores available in them, thus achieving optimal resource utilization. Early-exit networks result in violating this fundamental requirement. To get good bang for the buck with beefier GPUs, we need

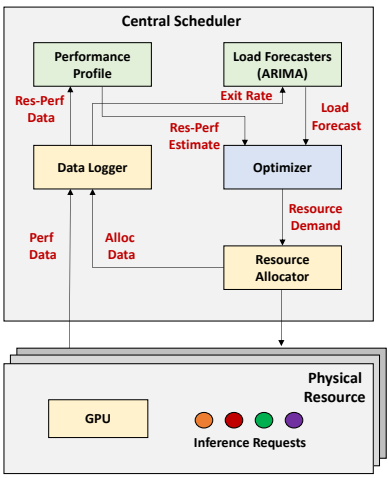


Figure 2: TETRIS architecture. Arrows indicate data flow directions and red text highlights the objects flowing in.

to have a large batch of samples to leverage massive parallelism. Paradoxically, due to nature of EE-DNN—which prefers small batches, this does more harm than good. For example, running smaller batch size with 26 P100 instead of a large batch size with 16 V100s yields better throughput.

Existing early-exit network architectures impose the condition that for a batch to exit at a ramp, *all the inputs in the batch must exit*. This is due to the need for additional operations necessary to reform the batch after each sample exits, and the overhead associated with it. As the batch size increases, the probability of all of the samples in the batch exiting at the same ramp decreases exponentially. Thus, larger batches always negate the benefits of early exits. Even if this engineering limitation is circumvented, *early-exit networks result in significant underutilization of the GPUs*. This is because the inputs in a batch can exit at different points in the DNN, and thus, the size of the batch *shrinks* as the inference proceeds. Due to the shrinkage in batch size (fig. 1), the GPUs are not utilized fully, leading to poor throughput. As a result, existing early-exit networks have restricted the use of batching, negating their benefits.

3 TETRIS System

TETRIS seeks to mitigate the limitations of early-exit networks and utilize them to provide faster and resource efficient inference. The key idea TETRIS uses to achieve its goal is to *maintain a constant batch size* during the execution of the early-exit network. To do so, it splits a DNN model into parts, places each part on different GPUs, and then executes them in a pipelined fashion. Hence, TETRIS needs to determine the optimal number of splits, and the optimal number of GPUs to run the splits on. For the former, TETRIS utilizes an online profile estimation technique that is computationally light, and for the latter, it proposes a dynamic programming based optimization coupled with a heterogeneity-aware model-parallel execution. We describe them in detail in this section. The overall architecture of TETRIS is depicted in fig. 2.

3.1 Online Batch Profile Estimation

In order to be generally applicable to several early-exit architectures, TETRIS treats the early-exit network as a *black-box*. In particular, it does not assume any knowledge or make any assumptions about the early-exit mechanism or the model. The only requirement for TETRIS is that it able to query the batch size at every exit ramp. As we show later, relaxing this assumption a little by assuming that the model provides a way to *disable* an exit ramp (e.g., by a REST interface) improves TETRIS's performance (§ 3.4), but it is not a requirement.

TETRIS makes the determination of the optimal number of splits for an early-exit DNN based on the batch size reduction characteristics. For this purpose, it needs to determine how batch size changes over the course of execution of the EE-DNN model. Since the inference workloads is time-varying [11], TETRIS needs to do this determination in an online fashion. For this purpose, we assume that there is short-term predictability in the workload under consideration. We believe this to be a reasonable assumption; based on our conversations with the administrators of a large-scale inference pipeline, the workload variations are at large timescales, in the order of several hours. TETRIS uses ARIMA [13], a time-series forecasting method to determine the batch profile for a early-exit DNN.

We divide the workload into chunks of 2 minute intervals, and use a sliding window over the workload requests to prepare the input timeseries for the online profiler. In each window, the input to the profiler is the batch size at each of the exit ramps in the EE-DNN model. An example is depicted in fig. 1, where the EE-DNN has many exit ramps and each of the ramps (corresponding to each layer in the model), and each exit is annotated with the batch size. The model ingests inputs at a batch size of 16. The estimator outputs its forecast of the expected batch sizes at each of the exit ramps in a rolling fashion. Due to the time-varying nature of the workload, the estimator runs continuously, and we show the performance and efficacy of TETRIS's online batch profile estimator in §5.

We note that the use of a timer-based batch profile estimation results in cause issues with sudden spikes in the workload. In such cases, the effect of sudden spikes/workload dynamism is similar to traditional inference pipelines and thus, existing techniques for sudden workload variations are applicable to TETRIS. We incorporate a slack for the SLO in TETRIS's optimizer (§ 3.2), and TETRIS can use buffer resources if available. In its current design, TETRIS drops requests that cannot be served, similar to Clockwork [8]. Since TETRIS monitors its estimated batch profile compared to observed (fig. 2), it can reactively re-run the optimizer if they differ drastically. We defer the exploration of proactive workload spike mitigation techniques to a future work.

3.2 Dynamic Programming based Optimization

The objective of TETRIS is to maintain the batch size nearly constant during the execution of the EE-DNN. In our earlier example of a EE-DNN with exit ramps (fig. 1), one solution to maintain the batch size constant is to split the model into two parts. For instance, one may slice at the end of the exit ramp where the batch size shrinks to 8, thus creating two splits of the model—the first split ends with the ramp where the batch size shrinks to 8, the second split contains the rest of the model. The split model can then be executed in the following fashion: we execute the first split twice (consuming two batches of 16 inputs), resulting in two outputs of batch size 8 each; then we combine the two outputs to obtain a batch size of 16 for the second split. While this maintains the batch size to 16 throughout the execution of the EE-DNN, in general, we need to account for the execution time, the latency constraints on the inputs and several other criteria. Thus, TETRIS formulates the splitting and execution of the EE-DNN model as an optimization problem.

Consider the task of executing a EE-DNN model with L layers for a workload under consideration with a latency constraint of *SLO* ms and request rate of R queries per second. TETRIS's goal is to cut the EE-DNN model into the optimal number of splits. For a particular split of the model with N layers in it, we can define the execution time or *cycle time* as:

$$CycleTime = A(0 \rightarrow N, B_{0 \rightarrow N}) \tag{1}$$

Since the request rate is R, we can estimate the largest batch size, B_0, that is possible that does not violate the SLA. Using these definitions, the throughput of the system can be computed as

$$Throughput = \frac{B_0}{CycleTime} \tag{2}$$

and the worst case latency, $Latency_{wc}$ is simply $CycleTime$. Our aim is to satisfy the following constraints:

$$Latency_{wc} \leq SLO - Slack$$
$$Throughput \geqslant Throughput_{baseline}$$
$$Cost \leq \alpha \times Cost_{baseline}$$

where *Slack* is the allowed slack in SLA ($\geqslant 0$), *baseline* is the baseline DNN model and α is a cost multiplier. We can then define a dynamic programming based recursive optimization:

$$A(i \rightarrow j, B_{i \rightarrow j}) = \min_{i \leqslant s \leqslant j} \begin{cases} A(i \rightarrow s, B_{i \rightarrow s}) + \\ T(s+1 \rightarrow j, B_{s+1 \rightarrow j}) \end{cases}$$

where $T(i \rightarrow j, B_{i \rightarrow j}) = \sum_{k=i}^{j} P(k, B_k)$. In this formulation, P is the throughput-latency profile, $B_{0 \rightarrow N}$ is the estimated batch profile for the EE-DNN model with N layers, B_k is the estimated batch size at layer k, and B_0 is the maximum batch size that can be supported, derived using the request rate R. The solution to this optimization formulation gives the optimal splits for the EE-DNN model.

3.2.1 Leveraging Model Parallelism

In the above formulation, the splits of the EE-DNN model are executed in the same GPU in a serial fashion. This may be the optimal settings in some cases, but the availability of many GPUs in a cluster provides TETRIS with the opportunity to execute the splits in parallel, commonly referred to as *inter-layer model parallelism*. In typical large scale inference scenarios, model parallelism has shown to perform worse compared to data parallelism due to the additional communication overhead incurred by it. While this is true, TETRIS can account for it as an additional constraint in its formulation.

If there are m machines available in the cluster, then we can modify TETRIS's optimization formulation as:

$$A(i \rightarrow j, m, B_{i \rightarrow j}) = \min_{i \leqslant s \leqslant j} \min_{1 \leqslant m' < m} \begin{cases} A(i \rightarrow s, m - m', B_{i \rightarrow s}) + \\ T_x(s, s+1) + \\ T(s+1 \rightarrow j, m', B_{s+1 \rightarrow j}) \end{cases}$$

where $T(i \rightarrow j, m, B_{i \rightarrow j}) = \sum_{k=i}^{j} P(k, m, B_k)$. T_x is the communication time for transferring data from the end of a split to the next and each GPU (machine) processes $\frac{B_k}{m}$ samples. In addition to minimizing the number of splits, the formulation also tries to minimize the resources to run the splits.

3.2.2 Pipelining

Due to the use of model parallelism, TETRIS may incur GPU under-utilization if the communication costs dominate. To mitigate this, we adopt a simple pipelining strategy. Each GPU processing a split can simply process the next batch once it is done with the current batch. This allows the GPU to overlap computation and communication. In the steady state of such a pipeline, TETRIS's optimization can be modified to optimize $A(i \rightarrow j, m, B_{i \rightarrow j})$ as:

$$\min_{i \leqslant s \leqslant j} \min_{1 \leqslant m' < m} \max \begin{cases} A(i \rightarrow s, m - m', B_{i \rightarrow s}) \\ T_x(s, s+1) \\ T(s+1 \rightarrow j, m', B_{s+1 \rightarrow j}) \end{cases}$$

where the pipelining is able to hide the latency from sum of all parts to the maximum latency incurred by any one.

3.2.3 Accommodating Heterogeneity

TETRIS is further able to exploit heterogeneity in the hardware configuration, if available, to its advantage. Since GPUs differ in their computational capabilities and cost, having a mix of GPUs can be beneficial in TETRIS's model parallel execution strategy. For instance, each split can have different computational requirements, and placing the split on the right

$$A(i \to j, m, B_{i\to j}) = \min_{i \leqslant s \leqslant j} \min_{c \in C} \min_{1 \leqslant m' < mc} \max \begin{cases} A(i \to s, mc - m', B_{i\to s}) \\ T_x(s, s+1) \\ T(s+1 \to j, c, m', B_{s+1\to j}) \end{cases}$$

$$T(i \to j, c, m, B_{i\to j}) = \sum_{k=i}^{j} P(k, c, m, B_k)$$

where:

P is the throughput-latency profile for GPU config c
$B_{0\to N}$ is the estimated batch profile for EE-DNN model with N layers
B_k is the estimated batch size at layer k; each GPU processes B_k/m samples
B_0 is estimated using R, request rate
mc is number of GPUs of configuration c in data-parallel mode
C is the set of GPU configurations available

Figure 3: The optimization formulation in TETRIS

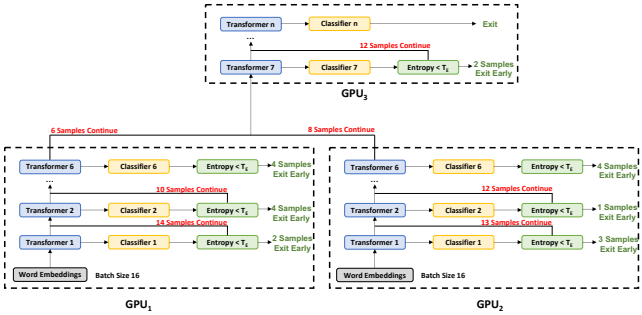


Figure 4: TETRIS uses a model-parallel execution strategy to run the splits of the EE-DNN model on heterogeneous hardware.

hardware configuration can both reduce cost and improve utilization. Towards this, TETRIS incorporates heterogeneity in its optimization formulation by accounting for the configuration of the GPUs available.

Figure 3 shows TETRIS's final optimization formulation.

3.3 Heterogeneity Aware Model-Parallel Execution

TETRIS's optimizer results in the apt number of splits for the EE-DNN model, the number of (heterogeneous) resources to place them, and the batch sizes to run the splits with. TETRIS uses a heterogeneity-aware scheduler to execute them.

The scheduler manages all the resources available in the cluster and uses a lightweight mechanism to probe the worker machines for their availability. Since DNN inference is highly predictable [8], the scheduler knows exactly the amount of time necessary to execute each split. Using the output from the optimizer, TETRIS's scheduler places the split in the available resources and starts the model parallel execution. The input is batched to attain the correct batch size and directed to the machines hosting the model splits. When a split has finished execution, the outputs are then directed to the machines hosting the next split, where multiple batches are fused to bring the batch to the correct size. The scheduler provides constant feedback to the optimizer on the availability of the machines for the next prediction period.

3.3.1 Pipelining

Since TETRIS uses model-parallel mode of execution, it incurs communication overheads where the outputs from one split of the EE-DNN model needs to be communicated to the next split. This may lead to GPUs being idle when the communication happens, so TETRIS incorporates a simple pipelining scheme to overlap computation and communication.

Each split independently executes batches, and upon completion of the batch, immediately moves on to the next batch. The machine hosting the next split maintains a queue that holds the partial results until it has received such inputs from all the machines. A potential problem in this pipelined execution strategy is if the split execution times are not balanced—the queues may build up and result in SLO misses. TETRIS's optimizer considers the execution time of the splits when determining the splits and the machines to host them on, hence under normal circumstances, such problems are not expected to arise. However, it is possible for some GPUs to become stragglers [10]. To combat such issues, TETRIS's scheduler maintains simple monitoring mechanisms to oversee the execution time of the splits on each of the resources, and marks stragglers to be excluded in the next assignment.

3.4 Improving TETRIS Further by Opening the Black Box of EE-DNNs

All the techniques we have outlined so far have assumed the EE-DNN to be a blackbox. Here, we outline how TETRIS can further be improved if this assumption is relaxed.

As we discuss in §2, one of the challenges in EE-DNNs is the overhead associated with exit checking. While TETRIS's model parallel execution reduces this overhead, it can be further mitigated by having more control over the EE-DNN. By providing information about the exit strategy to TETRIS, it can *control* the overheads and reduce it to a minimum. To do so, TETRIS provides a simple wrapper function, `exit-wrapper` that a developer of a EE-DNN would wrap the exit checking logic with. TETRIS uses this wrapper to control the exit logic's execution depending on the EE-DNN architecture (§2).

For EE-DNNs where each exit is *independent*, i.e., a decision to exit at a ramp is made just by the logic at that particular ramp, TETRIS simply disables all the ramps in a split other than at the end of the split. This is intuitive, because those exits serve no purpose in TETRIS's execution model. For EE-DNN architectures where exits are dependent, i.e., the decision to exit at a ramp is made using information from earlier ramps, TETRIS keeps track of this information to determine whether the logic has to be executed within a split. Currently, TETRIS supports three types of EE-DNN architectures which captures a large fraction of EE-DNNs. We reiterate that using the wrapper is not a necessity; rather, it simply provides opportunity for TETRIS to achieve even better performance.

We see further avenues to improve TETRIS's benefits by providing more control over the EE-DNN model. As an example, if TETRIS is able to control the entropy checking logic along with the exit check logic, say by using an API, then it can provide granular control to the user. For instance, it can let the user traverse the accuracy-latency curve in a fine-grained manner by dynamically adjusting the entropy and exit determination logic, depending on the workload and the user input. Additionally, it can also dynamically enable and disable exits in an online fashion, say by employing a multi-armed bandit algorithm that uses the current workload to determine which exits are useful. This can augment TETRIS's online batch profiler to provide further benefits. We plan to explore these directions in a future work.

4 Implementation

The TETRIS scheduler and client are implemented in 2800 lines of python code. Our prototype currently supports DNNs written in PyTorch and supports inference on GPUs. At runtime, the query scheduler accepts queries using a REST API. Resource reallocation events are triggered by a timer-based event, which is raised every 2 minutes in our experiments. A 2 minute allocation window is long enough for the allocation to reach its steady state that performance metrics from the job would be reliable while at the same time frequent enough to adapt to changes in the load and learned performances.

5 Evaluation

We evaluate TETRIS using a variety of workloads and compare it against both state-of-the-art (SOTA) EE-DNN models and stock DNN models. Our key results show that:

- For a fixed set of resources, TETRIS is able to provide up to 70% and 74% better goodput compared to SOTA EE-DNN models in NLP and computer vision, respectively. TETRIS also outperforms stock DNN models by up to 32% and 58%. Additionally, TETRIS's improvement increases with increase in batching opportunities or availability of heterogeneous resources. (§ 5.1)
- When the performance requirements, such as the desired throughput, is fixed, TETRIS is able to achieve them at substantially lower cost: TETRIS incurs 35% to 78% lower cost depending on the batching opportunities. (§ 5.2)

Experimental Setup: We run our experiments in a number of different GPUs. The GPUs we use are NVIDIA's V100, P100 and K80. Our largest cluster size was 39 (§ 5.7.4), comparable to recent work [8]. Each server has one 12-core Intel Xeon E5-2690v4 CPU, 441 GB of RAM, and one or more NVIDIA GPUs. GPUs on same server are interconnected via a shared PCIe interconnect, and servers in the cluster are interconnected via a 10 Gbps Ethernet interface. All servers run 64-bit Ubuntu 16.04 with CUDA library v10.2 and PyTorch v1.6.0.

Datasets: We use ResNet-50 from TorchVision and BERT-Base, BERT-Large from Transformers for experimentation. Following previous work, we run ImageNet and the GLUE benchmark in closed-loop clients. We run multiple experiments, varying batch sizes from 1 to 64.

Comparison & Metrics: We compare TETRIS against DeeBERT [29] and BranchyNet [27], two representative EE-DNNs in NLP and vision domains, respectively. For the majority of our experiments, we focus on the NLP model, because they form the bulk of industry workloads such as search and recommendation. Each EE-DNN can be tuned to a specific early exit entropy, which determines the tolerable error. Unless otherwise specified, we pick the entropy to be 0.4, which results in less than 2% error in our experience. We pick this value because it also matches with the accuracy loss incurred by DistilBERT [22], a distilled version of the BERT model. Note that setting the entropy value to be too low negates the usefulness of early-exits (§2).

Our main metric of comparison is *goodput*, or the number of samples per second that can be sustained without violating the SLO. In all our experiments, unless stated, we use an SLO of 100ms. We found this to be close to the value used by real-world workloads through conversations with our industry partners, and also in the range of values used by recent works [8]. We use batching by default, and set different batch sizes ranging from 2 to 16.

5.1 Overall Performance

We first present the overall picture. To do so, we compare TETRIS against the comparisons, for varying batch sizes. We assume that the cost to be constant, i.e., both TETRIS and comparison systems use resources that cost the same.

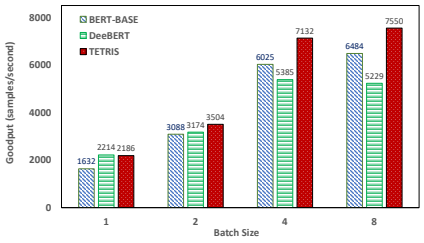


Figure 5: TETRIS is able to outperform NLP EE-DNN models by up to 44% in homogeneous settings.

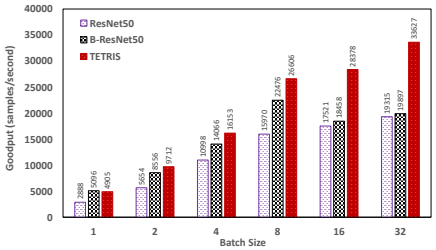


Figure 6: TETRIS is able to outperform vision EE-DNN models by up to 74% in homogeneous settings.

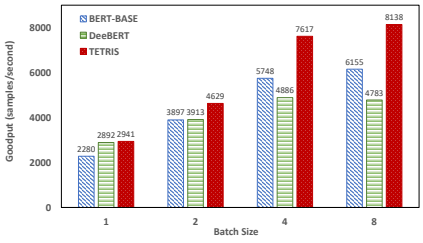


Figure 7: TETRIS is able to outperform NLP models by up to 70% by leveraging heterogeneous resources.

In figs. 5 and 6, we depict the performance of TETRIS when the cluster consists of *homogeneous* resources. We conduct this experiment in a cluster of 16 NVIDIA V100 GPUs, hence both TETRIS and the comparison systems use all the 16 GPUs. As we see, when the batch size is 1, EE-DNN is able to outperform the BERT model. This is expected, as the EE-DNN is able to "exit" many of the samples early. However, as the batch size increases, we see that the EE-DNN model becomes progressively worse compared to the non-EE model, BERT-BASE, which is now able to utilize the massive parallelism offered by the GPU. TETRIS on the other hand, is able to outperform BERT-BASE in all cases, and DeeBERT in all cases except when the batch size is 1. When the batch size is 1, TETRIS incurs a small penalty due to its model-parallel execution. TETRIS's performance improvement increases with increase in batch size, and is able to provide up to 44% increase in goodput compared to DeeBERT, and up to 30% compared to BERT-BASE.Note that even at batch size of 8, the models are not able to saturate the GPU. The improvements are bigger in vision models, where TETRIS is able to provide up to 74% better goodput. Figures 7 and 8 show the performance of TETRIS when the cluster consists of *heterogeneous* resources. Here, our cluster consists of a mixture of NVIDIA V100, P100 and K80 GPUs. Since we maintain the cost to be constant, we pick the configuration (type and number) of GPUs for each of the systems that maximizes the goodput. For instance, since the early-exit models are unable to support larger batch sizes, and thus not able to leverage the parallelism in the GPU, it is almost always better to allocate cheaper GPUs. On the other hand, the non early-exit models are always better using the most capable GPUs as long as there are enough opportunities for batching. Thus, neither are able to exploit the heterogeneity. In contrast, we see that TETRIS is able to effectively utilize the different GPUs and outperform the comparisons. For each batch size, TETRIS's profiler and optimizer is able to identify the optimal configuration that maximizes the goodput (in § 5.7.4 we investigate the effect of heterogeneity in depth). Here, TETRIS's techniques provide up to 70% improvement.

5.2 Cost Effectiveness

We focused on the ability of TETRIS to provide better goodput by maintaining the cost constant in the previous experiment. In this experiment, we evaluate the ability of TETRIS to reduce the cost of inference when the throughput is fixed. To do so, we fix the desired throughput to be 6000 samples per second. We then consider two settings: in a homogeneous cluster scenario which consists of NVIDIA V100 GPUs, we determine the number of GPUs that are necessary to sustain the desired performance for all the models; and in a heterogeneous cluster scenario consisting of V100, P100 and K80 GPUs, we determine the minimum cost incurred to sustain the desired performance. Note that since the pricing of the GPUs vary drastically between service providers, we use the average price as a rough indicator of the current price. The results are shown in figs. 9 and 10 respectively.

We notice that TETRIS provides the best performance in all settings and all batch sizes. When the batch sizes are small, none of the models are able to utilize the GPUs efficiently, resulting in the need to use more GPUs. As the batching opportunities increase, both BERT and TETRIS are able to utilize the resources better, and hence the number of resources required reduces. DeeBERT is also able to use batching to reduce the resources, but due to the batching limitations of EE-DNNs, the resource utilization is not as efficient as TETRIS or BERT, resulting in it using more resources than either of them. TETRIS is also able to provide the best performance at the lowest cost. Again, we see that at lower batch sizes, the number of resources required to sustain the performance is higher and that drives the price higher even with heterogeneity. When this is not the case, TETRIS is able to provide better performance per dollar compared to its comparisons. Here, TETRIS is able to provide the same performance at 35 to 78% lower cost depending on the batch size.

5.3 Workload Adaptability

Here, we seek to answer the question ***"Can* TETRIS *adapt to workload variations?"*** by evaluating its batch profiler and optimizer. To do so, we created three variations of the workloads by varying the ratio of the easy and hard examples. The

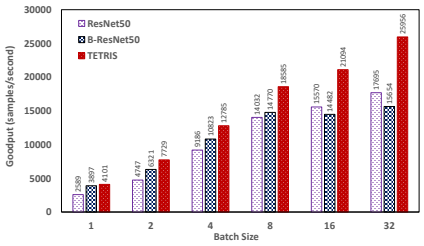


Figure 8: Using heterogeneous resources boosts TETRIS's performance on vision models.

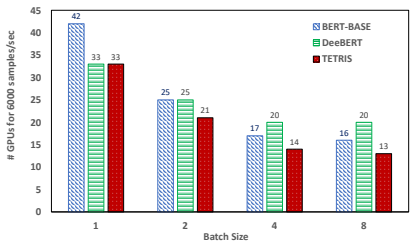


Figure 9: When the performance requirements are fixed, TETRIS is able to achieve it using less resources.

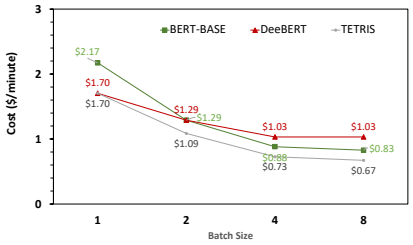


Figure 10: TETRIS incurs the lowest cost to achieve a given performance requirement, and reduce the cost of inference by up to 35%.

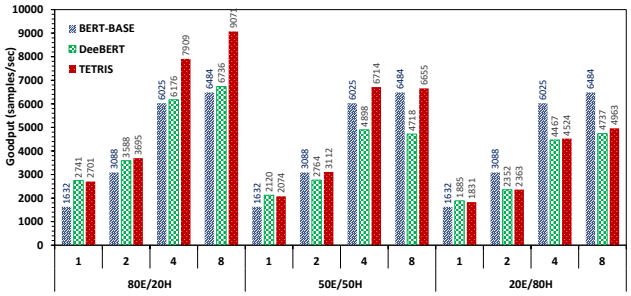
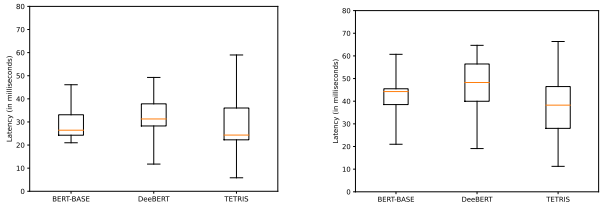


Figure 11: TETRIS's online batch profiling (§ 3.1) and optimizer (§ 3.2) are able to adapt to workload variations.



(a) Homogeneous **(b)** Heterogneous

Figure 12: Latency Distribution

easy to hard ratio were fixed to be 80:20, 50:50 and 20:80 in the three workloads, respectively. We then ran the inference in closed loop on each of the models on both homogeneous and heterogeneous resources, switching between the workloads at fixed intervals. That is, at the beginning of the experiment, we start with the 80:20 mix, and after a specific time, we switch to the 50:50, and then to the 20:80. Figure 11 shows the results.

We notice that the EE-DNNs provide benefits compared to the non early-exit models when the batch size is small and the difficulty is skewed towards easy examples. This is expected, because the easy examples are able to leverage the early exit mechanism. However, as the batch size increases, or when the difficulty changes, EE-DNNs become worse as they are unable to take advantage of the GPU parallelism (as in the previous experiment), or the exit checking overheads accumulate. The non early-exit models perform poorly when the workload consists of a large fraction of easy inputs, but are able to provide good performance when the batch sizes are large, or when the inputs are mostly hard, requiring the entire predictive power of the model.

In contrast, TETRIS is able to effectively adapt to the requirements of the workload. When the workload is skewed towards easy inputs, TETRIS behaves like a EE-DNN model. The profiler is able to capture the hardness quickly, and the optimizer is able to split the model to achieve good performance, regardless of the batch size. When the workload becomes majorly hard, the profiler identifies this again and based on this identification, TETRIS's optimizer is able to suggest the most optimal split based on the hardness. Thus, TETRIS behaves similar to the non early-exit model in this case, with the added benefit of being able to leverage heterogeneity through its model-parallelism based execution.

5.4 Latency Implications

Since TETRIS depends on a split-execution model, where parts of a model may be executed on different GPUs, it is natural to assume that the benefits of TETRIS comes at the cost of increased latency. To evaluate the implications of latency, we conducted an experiment where we measure the latencies incurred by TETRIS, BERT and DeeBERT over 100K inferences. Figure 12 shows the median, quartiles, min and max latencies incurred by the three techniques in homogeneous and heterogeneous settings. The workload mix comprised of easy and hard examples, their ratio was fixed to be 50:50 and batch-size was set to 8.

TETRIS attains the lowest min, median, 25th-percentile and 75th-percentile latencies across the board, which may seem counter-intuitive. While TETRIS would incur additional latency compared to a non-EE model, this additional latency is incurred only by a fraction of the inputs. In contrast, in the non-EE model, every input incurs the same latency. Typically, only the hard inputs incur this penalty in TETRIS, which affects the tail (max) latency. Even then, the SLO is not violated, as TETRIS's optimizer considers the workload's current hardness ratio using its online batch profile (§ 3.1) and the network overhead in determining the splits (§ 3.2). When the workload is dominantly hard, TETRIS avoids splitting the

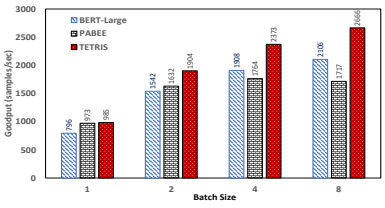


Figure 13: TETRIS's techniques are general and applicable to many EE-DNN architectures. It is able to boost the goodput of PABEE by up to 55%.

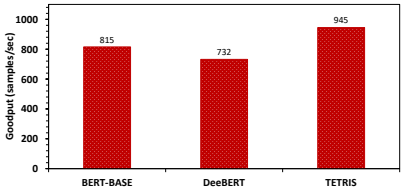


Figure 14: TETRIS is able to maintain its performance on real-world workloads, where the request arrivals are non-uniform in nature.

Model	Overhead (s)	
	Homogeneous	Heterogeneous
ResNet50	1.13	2.62
BERT-Base	0.87	2.09
BERT-Large	1.53	3.63

Figure 15: TETRIS incurs minimal overheads, its optimizer is able to find the optimal split, resources and batch size for a EE-DNN model in a few seconds.

model thus making it perform equivalent to the EE-model (fig. 11).

5.5 Generality

We try to answer ***"How general are the techniques in TETRIS?"***. For this, we apply the techniques in TETRIS on PABEE [31], an EE-DNN model that uses a sophisticated counter based mechanism to decide on the exit choice. This model represents a different architecture compared to DeeBERT. We repeat the experiment setup in § 5.1, and show the result in fig. 13. We see that TETRIS is able to provide upto 55% higher goodput compared to PABEE due to its consideration of EE-DNN architectures as black-boxes.

We note that EE-DNNs are orthogonal to model compression. Hence, the techniques we propose in this work can apply to pruned, distilled and quantized models for further improvements. Adding early exits to distilled model is outside the scope of this work, and due to the lack of availability of such an open-source model, we defer it to future work. Though compressed models can significantly improve inference latencies, they do so at the cost of significant accuracy reduction, accuracy trade-off shown in [17, 30, 32].

5.6 TETRIS on Realistic Workloads

Finally, we investigate whether TETRIS can maintain its benefits in realistic workloads. To evaluate this, we use the request arrival rate from the openly available Twitter trace [16] which is bursty with unexpected load spikes, scaled to have an average request rate of 1000 req/sec. We depict the goodput achieved by TETRIS and its comparisons in fig. 14. TETRIS is able to maintain its performance even when the arrivals are non-uniform. In this experiment, TETRIS attains 29% improvement in goodput over DeeBERT, and 16% improvement in goodput over BERT-BASE.

5.7 Microbenchmarks

5.7.1 Overheads

This experiment evaluates the overhead of TETRIS's optimizer. Since the optimizer uses a dynamic programming based solution to determine the optimal number of splits and the GPUs on which the splits should be run, we investigate if the optimizer could become an overhead. To do so, we measure the time taken for the optimizer to provide an output, as the number of variables change. Specifically, we change the number of GPUs available to TETRIS, and the number of layers in the EE-DNN model. We show the results of this experiment in fig. 15. As we see, the optimizer is light-weight, and is able to provide the optimal configuration within a few seconds.

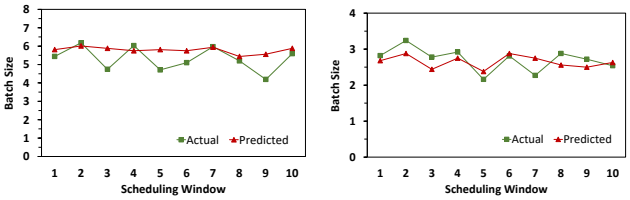


Figure 16: TETRIS's batch profile estimation closely matches reality.

5.7.2 Efficacy of TETRIS's Batch Profile Estimation

TETRIS depends on its online batch estimation (§ 3.1) to determine the splits of the model. To evaluate the efficacy of this technique, we conducted the following experiment. We place two cut points on the model (based on the workload), and estimate the batch size at these cut points at the beginning of every two minutes windows for an input batch size of 8. We then compared the average batch size seen during the two minutes against our prediction. Figure 16 shows the predicted and the actual batch sizes on the two cuts for 10 such windows. We observe that TETRIS's prediction closely matches reality. We note that perfect prediction is not necessary due to two reasons. First, since the batch profile estimation is simply a guidance for TETRIS's optimizer for determining the cut points, a rough estimate is enough. Second, when TETRIS actually executes the inference in a split fashion, its optimizer accounts for a slack in the SLO (§ 3.2) which we set to 20%. Thus, TETRIS is able to merge and combine batches at cuts as long as the variance is within the slack.

5.7.3 Impact of Error Tolerance

The advantage of EE-DNNs is the fact that they are able to provide the user with the ability to traverse the accuracy-latency curve in a smooth fashion instead of forcing them to pick a fixed point (§2). Here, we investigate how the error tolerance

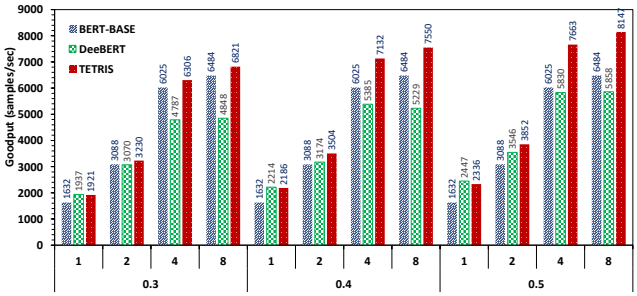


Figure 17: As the error tolerance increases, TETRIS is able to significantly improve its (already good) performance.

affects TETRIS's ability to provide benefits by varying the allowable error. Recall that the entropy value determines the error, hence we vary the exit entropy from 0.3 to 0.5, and show the results of the experiment in fig. 17. We note that tolerances outside these ranges are typically not useful. At low entropy values, none of the inputs are allowed to exit even if they could have. This makes early exits not useful. On the other hand, at high entropy values, all the inputs exit early, but at the cost of incurring a higher error. As we see, TETRIS is able to identify this, and tune the splits accordingly. If the user is willing to afford more errors, TETRIS is able to provide better goodputs, up to 43% higher compared to DeeBERT.

5.7.4 Impact of Heterogeneity

A key advantage to TETRIS is its ability to use heterogeneous hardware. This experiment investigates the effect of heterogeneity in TETRIS's performance. To conduct this experiment, we picked two configurations of machines that costs the same: a homogeneous cluster of 16 V100 GPUs, and a heterogeneous cluster of 6 V100, 8 P100 and 15 K80 GPUs. Both clusters cost $0.013 per second. We run the workload in these two clusters and report the goodput in fig. 18. The result shows that TETRIS is able to improve its performance by up to 34% by embracing heterogeneity. The improvement is best when the batch sizes are small, and the performance gap reduces as the batch size increases. This points to the usefulness of TETRIS's optimizer: it is able to split the EE-DNN is such a way to use many less powerful GPUs for the earlier splits and then combine the outputs on a more powerful GPUs.

5.7.5 Impact of SLO

Next, we evaluate the impact of SLOs on TETRIS's performance. Inference workloads are typically bounded by SLO guarantees that need to be strictly adhered. The SLO determines the maximum batch size that can be created; a strict SLO translates to less batching possibilities and hence smaller batches, and vice-versa. We consider SLOs from 25ms to 1000ms and translate it to the maximum batch sizes that can be supported. We assume sufficiently large request rate to create these batches. For each of such maximum batch sizes obtained by the SLO, we run closed loop inference with TETRIS and its comparisons. The result is shown in fig. 19.

When the SLO is very small, batching opportunities are virtually nil. At small batch sizes, DeeBERT (and EE-DNNs in general) offer compelling advantage over BERT. TETRIS is able to determine this and its optimizer is able to adapt; at a batch size of 1, TETRIS's goodput is just 1% lower compared to DeeBERT. However, as batching opportunities arise, both TETRIS and BERT are able to leverage the parallelism offered by the GPU and perform better. Here, TETRIS provides up to 63% (34%) higher goodput compared to DeeBERT (BERT).

5.7.6 Impact of Relaxing TETRIS's Assumptions

TETRIS makes no assumptions about the EE-DNN architecture to be general. However, as we describe in § 3.4, relaxing the assumptions can be beneficial. In this experiment, we assume that TETRIS is able to disable the exits which are not useful. The results in fig. 20 show that TETRIS is able to effectively mitigate the overheads of exit checking when they're not useful and improve its performance by up to 16%.

5.7.7 Impact of Model Parallelism

Here we investigate the usefulness of model parallel execution. To do so, we execute inference using TETRIS where we turn off model parallelism. When model parallelism is turned off, TETRIS is forced to execute the splits of the model in the same GPU in a serial fashion, waiting for all copies of a split to finish before it can start executing the next split. Figure 21 shows that the ability to execute the splits across GPUs in parallel significantly improves TETRIS's performance.

5.8 Shortcomings

There are two shortcomings of TETRIS. First, TETRIS is designed for workloads where there are enough opportunities to batch the input. When this opportunity ceases to exist, TETRIS does not provide benefits. Figures 5 and 6 show that for batch size of 1, TETRIS is up to 3% worse compared to the EE-DNN model. Our experience interacting with industry partners indicate that small batches are rare in the real-world, hence we believe TETRIS to be useful in a majority of cases. Second, EE-DNNs are built on the assumption that the workload consists of a mix of easy and hard examples. When the workload is predominantly hard, TETRIS is unable to find optimal splits or batching opportunities for its model parallel model. Figure 11 shows that TETRIS is up to 23% worse compared to the non-EE model when the workload is 80% hard. Note that distilled models are likely to be not useful in such cases, as they would incur significant accuracy loss. TETRIS is still better than EE-DNN, and by allowing more error tolerance, it is possible to boost its performance. However, such settings violate the basic assumptions of EE-DNNs, so we do not consider them to be a good usecase.

6 Related Work

Deep learning systems have recently received significant attention, and a number of work has focused on improving the performance of deep learning *training* [14, 20]. In addition

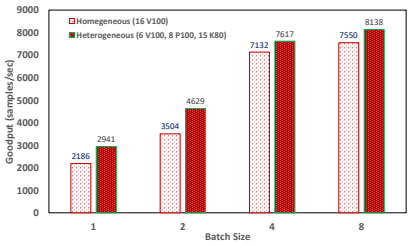


Figure 18: Heterogeneity, especially when batch sizes are small, aids TETRIS improve its performance by up to 34%.

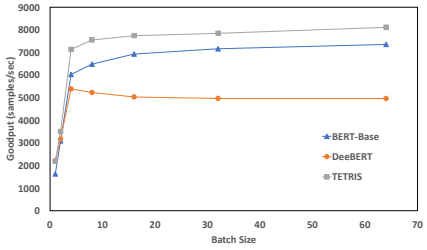


Figure 19: As the opportunities for batching increases, TETRIS is able to improve its performance.

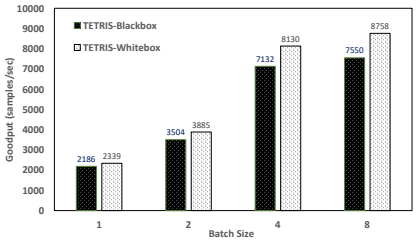


Figure 20: By relaxing the assumptions of treating the EE-DNN as a blackbox, TETRIS is able to improve its performance.

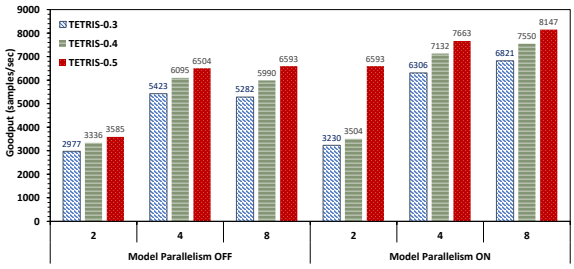


Figure 21: By using model parallelism, TETRIS is able to parallelize the execution of EE-DNN model splits across GPUs.

to the data parallelism and model parallelism provided by popular open-source frameworks such as PyTorch [1] and TensorFlow [2], recent works have proposed hybrid parallelism strategies. Further, pipelining and compression has been shown to boost training performance. However, the latency of inference is of critical importance, while training focuses on improving the throughput. Though the optimizations introduced for training can carry over, the underlying assumptions make inference face its own set of challenges.

Model serving systems are designed to maximize system throughput under strict latency constraints, often using model replicas. Prior studies have primarily focused on sophisticated cluster-level scheduling, placement, and co-ordination strategies for inference queries [4, 8, 9, 21, 24]. However, to the best of our knowledge, none of the existing works on inference systems focus on leveraging early-exit networks.

Researchers have proposed number of well-explored methods to accelerate model inference. **Pruning** [6, 7] removes redundant parts of the neural model, from individual weights to layers and blocks. **Quantization** [25] reduces the number of bits needed to operate a neural model and to store its weights. **Distillation** [12, 15] transfers knowledge from large teacher models to small student models. These methods typically require pre-training from scratch and produce only one *compressed* model. However, such "static" solutions cannot adaptively adjust their complexity during inference. Complexity (or "difficulty") of input samples varies in most real-world scenarios, and shallow layers are often capable of correctly identifying some "easy" inputs. Ideally, these "easy" samples should be identified at certain early exits without executing deeper layers, thus reducing the overall inference latency with minimal impact on accuracy.

Several **early exit networks** have been proposed to accelerate inference of BERT and similar multi-layer transformer models by leveraging varying input sample complexity. Instead of using only one classifier, additional classifiers are attached to each transformer layer, each acts as *off-ramp* allowing samples to exit early without passing through the entire model. Key question in early exit networks is the criteria used to decide whether to exit early or continue to the next (more expensive and more accurate) classifier. At inference time, if the *certainty level* is higher than a pre-defined threshold, sample performs early exiting. Previous studies have proposed various *heuristic* criteria to judge *certainty level*. *Confidence-based criterion* [18, 23, 28] interpret the label scores output by softmax as confidence scores. *Entropy-based criterion* [29, 30] rely on entropy of predicted probability distribution to be smaller than pre-defined threshold. *Counter-based criterion* [31] require off-ramps classifiers to continuously generate identical predictions for pre-defined times. *Voting-based criterion* [26], inspired from ensemble technique, requires pre-defined number of off-ramp classifiers to reach an agreement. Additionally, several approaches choose not to rely on heuristic criteria, and introduce an additional module which *learns-to-exit*. Often such a module is a simple one-layer fully-connected network, which is shared among all off-ramps and outputs the certainty level [17, 32]. As we show in this paper, early-exit networks suffer from fundamental challenges (§2). TETRIS overcomes these and leverages EE-DNNs to provide superior inference performance.

7 Conclusion

We presented TETRIS, the first system, to the best of our knowledge, that focuses on making early exit networks practical and leverage them for fast and efficient inference. We identified the inability of EE-DNNs to support batching as a major challenge in making them practical, and thus TETRIS incorporates several techniques to address this challenge. It uses an ARIMA based online batch profile estimation technique and uses the estimate to formulate the EE-DNN execution

task as a dynamic programming based optimization problem. Embracing model parallelism, TETRIS splits the EE-DNN model, places them on different (heterogeneous) GPUs, and executes them in a pipelined fashion to ensure that the batch size remains constant. Our evaluation shows that TETRIS is able to improve inference goodput and provide significant reduction in cost: it is able to provide up to 74% higher goodput and reduce the cost by up to 78% compared to state-of-the-art EE-DNN models in NLP and computer vision.

References

[1] PyTorch. https://pytorch.org/.

[2] TensorFlow. https://www.tensorflow.org/.

[3] Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language\-model-by-microsoft/.

[4] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*, 2020.

[7] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. Compressing BERT: Studying the effects of weight pruning on transfer learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*, pages 143–155, Online, July 2020. Association for Computational Linguistics.

[8] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[9] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: Leveraging Ensemble Learning for Optimized Model Serving in Public Cloud. *arXiv e-prints*, page arXiv:2106.05345, June 2021.

[10] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 418–430, 2019.

[11] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv e-prints*, page arXiv:1503.02531, March 2015.

[13] Siu Lau Ho and Min Xie. The use of arima models for reliability forecasting and analysis. *Computers & industrial engineering*, 35(1-2):213–216, 1998.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pages 103–112, 2019.

[15] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for Natural Language Understanding. *arXiv e-prints*, page arXiv:1909.10351, September 2019.

[16] Abeer Abdel Khaleq and Ilkyeun Ra. Cloud-based disaster management as a service: A microservice approach for hurricane twitter data analysis. In *2018 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–8, 2018.

[17] Kaiyuan Liao, Yi Zhang, Xuancheng Ren, Qi Su, Xu Sun, and Bin He. A global past-future early exit method for accelerating inference of pre-trained language models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2013–2023, Online, June 2021. Association for Computational Linguistics.

[18] Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. FastBERT: a self-distilling BERT with adaptive inference time. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6035–6044, Online, July 2020. Association for Computational Linguistics.

[19] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[22] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[23] Roy Schwartz, Gabriel Stanovsky, Swabha Swayamdipta, Jesse Dodge, and Noah A. Smith. The right tool for the job: Matching model and instance complexities. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6640–6651, Online, July 2020. Association for Computational Linguistics.

[24] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):8815–8821, Apr. 2020.

[26] Tianxiang Sun, Yunhua Zhou, Xiangyang Liu, Xinyu Zhang, Hao Jiang, Zhao Cao, Xuanjing Huang, and Xipeng Qiu. Early exiting with ensemble internal classifiers. *CoRR*, abs/2105.13792, 2021.

[27] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. *arXiv e-prints*, page arXiv:1709.01686, September 2017.

[28] Keli Xie, Siyuan Lu, Meiqi Wang, and Zhongfeng Wang. Elbert: Fast albert with confidence-window based early

exit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7713–7717, 2021.

[29] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. DeeBERT: Dynamic early exiting for accelerating BERT inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, Online, July 2020. Association for Computational Linguistics.

[30] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. BERxiT: Early exiting for BERT with better fine-tuning and extension to regression. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 91–104, Online, April 2021. Association for Computational Linguistics.

[31] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. In *Advances in Neural Information Processing Systems*, volume 33, pages 18330–18341. Curran Associates, Inc., 2020.

[32] Wei Zhu. LeeBERT: Learned early exit for BERT with cross-level optimization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2968–2980, Online, August 2021. Association for Computational Linguistics.