Improving DNN Inference Throughput Using Practical, Per-Input Compute Adaptation

Anand Iyer[†] Mingyu Guan[†] Yinwei Dai[§] Rui Pan[§] Swapnil Gandhi[*] Ravi Netravali[§]
[†]Georgia Institute of Technology [§]Princeton University [*]Stanford University

Abstract

Machine learning inference platforms continue to face high request rates and strict latency constraints. Existing solutions largely focus on compressing models to substantially lower compute costs (and time) with mild accuracy degradations. This paper explores an alternate (but complementary) technique that trades off accuracy and resource costs on a per-input granularity: early exit models, which selectively allow certain inputs to exit a model from an intermediate layer. Though intuitive, early exits face fundamental deployment challenges, largely owing to the effects that exiting inputs have on batch size (and resource utilization) throughout model execution. We present E^3, the first system that makes early exit models practical for realistic inference deployments. Our key insight is to split and replicate blocks of layers in models in a manner that maintains a constant batch size throughout execution, all the while accounting for resource requirements and communication overheads. Evaluations with NLP and vision models show that E^3 can deliver up to 1.74× improvement in goodput (for a fixed cost) or 1.78× reduction in cost (for a fixed goodput). Additionally, E^3's goodput wins generalize to autoregressive LLMs (2.8-3.8×) and compressed models (1.67×).

1 Introduction

Machine Learning (ML) inference, or the process of deploying trained models to serve queries, has become a dominant and critical workload that underlies many real-world applications [34, 54]. Indeed, industry-scale inference systems are already presented with trillions of queries per day (i.e., 1000s *per second*) [6]), with the numbers continuing to rise as ML-powered services grow in number and popularity. Coupled with the steadily increasing sizes of deep neural networks [12, 41, 42, 65, 72], compute overheads and limitations are a paramount concern for inference systems today.

Given the practical importance of inference workloads, numerous techniques have been developed to lower compute overheads for serving by *compressing* models (§2), e.g., distillation [36, 43], pruning [24, 27], and quantization [62]. For instance, DistilBERT [57], a distilled version of the popular language model BERT [23], is ≈40% smaller and 60% faster. Yet, compression alone is not enough. New paradigms such as auto-regressive generative models inflate compute by mandating multiple passes through a model per input. Moreover, as model sizes grow, so too do their compressed variants when needing to maintain acceptable accuracies [46].

This paper studies complementary pathways to further tame compute overheads in large-scale ML serving via finer-grained compression. More specifically, an approach that has recently garnered attention in the ML community is *early-exit networks* [32, 49, 50, 59, 68–70, 73] (EE-DNNs), which propose the idea that inputs to a DNN can exit at intermediate model layers, rather than having to strictly traverse the full model. Easy inputs can safely (accuracy-wise) exit early, while hard inputs can continue through to the end to leverage the full expressiveness of the original model. In contrast to the above techniques, EE-DNNs adapt computation on a *per input* basis, rather than only compressing models for all inputs. Put differently, EE-DNNs can reduce computation in a model (including compressed ones) to the lowest amount necessary to accurately respond to each input; coarser, model-level compression alone often performs more compute than necessary for many inputs (§2).

Despite their intuitive benefits, several key drawbacks (§2.3) have precluded the widespread deployment of EE-DNNs, including in our own large-scale production service where EEs afforded acceptable accuracy drops where compression could not (§2.4). First and foremost, EE-DNNs are fundamentally at odds with input *batching* – the predominant technique used to boost resource utilization and throughput for ML workloads [20, 29, 44, 61]. The issue is that the very technique that enables EE-DNNs to deliver throughput benefits – i.e., allowing inputs to exit at intermediate model layers – results in shrunken batches for later model layers (and thus, resource underutilization). Our results highlight how this inefficiency can lead to EE-DNNs degrading performance; it is for this reason that state-of-the-art early-exit systems disable the use of batching altogether, a non-starter for real-world deployment [49, 59, 68–70, 73]. Second, the exits that EE-DNNs add to the original models can impose non-negligible compute overheads, especially as model complexity grows and for inputs that must traverse most of the model.

We present E^3,[1] a system that makes EE-DNNs practical, and leverages them to enable high-throughput and cost-effective inference across diverse deployment settings. The key idea behind E^3 is simple: *maintain a constant batch size*

[1]for **E**fficient **E**arly-**E**xits.

SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1251-7/24/11.
https://doi.org/10.1145/3694715.3695978

throughout the execution of an EE-DNN. At a high level, E^3 accomplishes this by first identifying "splits" (i.e., contiguous blocks of layers) in EE-DNNs that are expected to yield constant batch size outputs, and then replicating certain splits so as to keep the overall batch size constant throughout the model. Yet, realizing this simple idea in realistic scenarios involves several challenges, which E^3 tackles using two key components.

First, to handle the variability in workloads that alter the usage and utility of each exit in an EE-DNN over time, E^3 proposes an online batch profiling estimation technique (§3.1). E^3's profiler is based on ARIMA [37] and characterizes how batch size shrinks as model execution progresses. This information guides E^3's splitting strategy, and also presents opportunities to deactivate unnecessary exits to keep compute overheads low. Importantly, although E^3's batch profiles closely match reality for our workloads, it only uses them as a guide and does not rely on perfect predictions (§5).

Second, based on the observation that realistic deployments contain multiple GPUs, E^3 incorporates an *inter-layer model parallel* scheduler that judiciously runs splits in a parallel fashion to best utilize the available resources. While model parallelism is typically reserved only for large models that fail to fit in a single GPU, E^3 uniquely shows how this paradigm can help address the EE-DNN batching challenge. The idea is that multiple GPUs grant E^3 additional flexibility for running splits (e.g., sequential vs. parallel), and wins from careful exiting can often outweigh (and alleviate) cross-GPU communication overheads.(§3.2.1)

Along these lines, E^3 formulates its model parallel scheduler as a Dynamic Programming (DP)-based optimization that takes as input a set of compute/network resources, an EE-DNN, and the output of E^3's batch profiler. E^3's scheduler then considers the resource needs, run time, and communication overheads imposed by each potential split to determine the optimal set of splits to use and the batch size for each that maximizes goodput subject to SLO constraints (§3.2). E^3 further reduces communication overheads by leveraging *pipelining* to overlap computation and communication across batches. Moreover, we highlight that the running of splits with different batch sizes presents new opportunities to favorably leverage *heterogeneous* hardware. We show how E^3's general DP formulation can be naturally extended to maximize throughput and cost reductions in such settings.

We implemented E^3 in PyTorch [8] (§4), and evaluated it across a variety of recent language and vision EE-DNNs and stock models, multiple workloads, and clusters with up to 46 GPUs. Across these scenarios, E^3 achieves up to 1.74× higher goodput for the same compute cost, or reduces the compute cost by up to 1.78× for a fixed goodput, all relative to state-of-the-art EE-DNNs. Importantly, E^3's goodput wins generalize to autoregressive LLMs (2.8-3.8×) and compressed models (1.67×). Lastly, our results show how E^3 empowers early exits to deliver on their potential benefits, enabling EE-DNNs to deliver 32-58% higher throughputs than their corresponding non-EE models. Crucially, despite their ramp overheads, EE-DNNs with E^3 yield comparable tail latencies, with substantial latency improvements at all other quartiles (§5).

2 Background

We start with a background on model compression techniques. We then describe the value that early exits bring to both uncompressed and compressed models, and the challenges (based on our production experience) associated with making EE-DNNs practical.

2.1 Model Compression Approaches

In striving for improved inference accuracy, ML models have steadily increased in complexity, with recent model versions incorporating more layers and parameters. For instance, BERT-LARGE has 340 million parameters, while GPT-2 and GPT-3 have 1.5 and 175 billion [17]. Unfortunately, even with the best accelerators, complex models are often incapable of satisfying the strict SLOs and high request rates seen in practice for user-facing applications [72]. Model compression has sought to resolve this problem by proposing techniques to replace the original, complex model with a simpler one without a significant reduction in accuracy. The insight behind compression is that only a fraction of the original model's predictive power is required for many inference tasks.

The most common compression techniques include pruning, quantization, and distillation [18, 27, 36, 62]; we discuss other optimizations in §6. Pruning is based on the notion that models are often over-parameterized. Hence, though computationally expensive [24, 27], identifying and removing the unnecessary parameters can result in a smaller model. In contrast, quantization reduces model size by employing lower-precision arithmetic; manipulating weights in this manner reduces the amount of storage necessary to house them. For instance, replacing 32-bit weights with a binarization process [62] can reduce model size by 32×.

Knowledge distillation [36, 43] has emerged as a popular compression technique in which 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 of the teacher. 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 its own pros and cons [28, 52].

Despite their promising benefits, all of these compression techniques leave compute reduction opportunities on the table by operating at a *model* level. Specifically, they fail to capitalize on unique opportunities that individual inputs bring, and instead apply compression to the full model in a way that caters to all inputs.

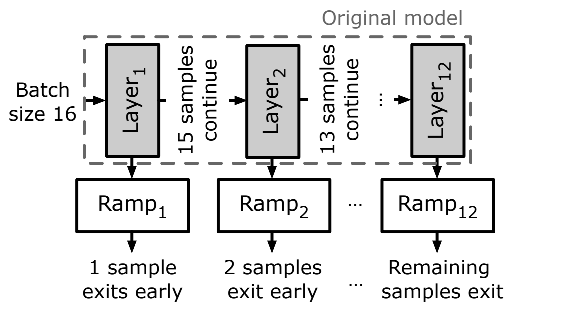

Figure 1. Early-exit DNNs allow inputs to exit after intermediate layers, thus reducing computation.

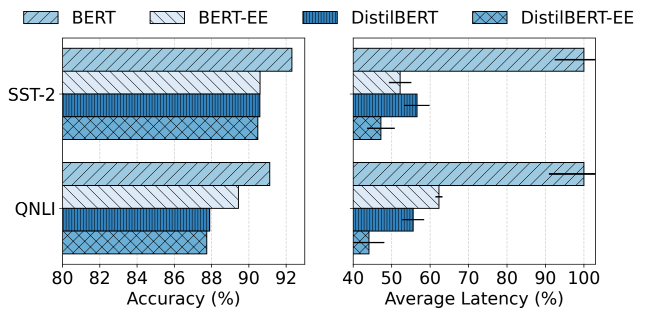

Figure 2. Early exits bring large compute (and latency) savings with only mild accuracy losses, including when running atop distilled models. All results use batch size 1, and latency results are normalized to those of vanilla BERT. Each bar shows the average with error bars spanning a standard deviation across five runs.

2.2 Early-Exit Networks

Early-exit networks (fig. 1) present a path-way for finer-grained compression, whereby computation overheads and accuracy are traded off using per-input decisions. More specifically, early-exit networks (EE-DNNs) are rooted in the idea that different inputs utilize the predictive power of a model to different degrees. The *hardness* of inputs in a workload varies, and EE-DNNs enable compute overheads to vary accordingly without substantial accuracy loss; hard inputs can use the original model's full predictive power (traversing all of its layers), while easy inputs may use only some layers before exiting with a prediction result. Importantly, since compute overheads are directly proportional to the number of layers executed in a model, exiting earlier translates to lower resource costs, higher throughput, and faster results.

An ideal early-exit network would, in theory, incur the optimal amount of computation for any given input. However, in practice, a decision to exit early has to be made at each exit point (often referred to as a *ramp*). ML researchers have proposed many forms of EE-DNNs [32, 49, 50, 59, 64, 68–70, 73] with various exiting techniques. The simplest ramp is an entropy computation that provides the confidence of the prediction at that point. More complex 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.

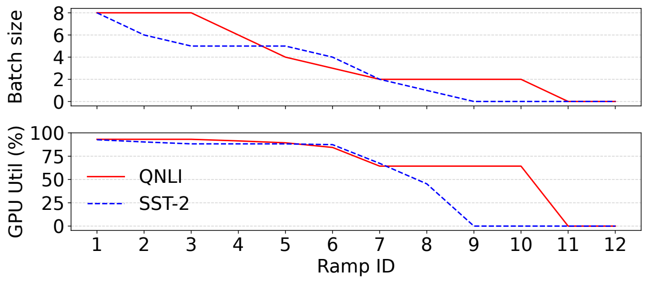

Figure 3. Samples in a batch exit DeeBERT [69] early as they pass through its ramps, which causes severe resource underutilization.

To better understand the compute savings and accuracy implications of early exits, we performed experiments using four variants of the popular BERT NLP model [23]: BERT, BERT-EE (a version of BERT that incorporates early exits [69]), DistilBERT (a distilled version of BERT [57]), and DistilBERT-EE (a version of DistilBERT that incorporates early exits). Unlike the other three variants, an off-the-shelf version of DistilBERT-EE has yet to be proposed. Thus, we developed it in house, using the same methodology that was employed to develop BERT-EE from BERT in [69], i.e., after each encoder block, adding an exit ramp with a bertpooler, a dropout layer, and a fully connected layer. Our evaluation considered two commonly used language datasets, SST-2 and QNLI [2]. Results (fig. 2) point to the following takeaways:

- **Early exits + Stock models:** adding exits to BERT resulted in average compute and latency savings of 42.7%, with minimal (1.7%) impact on accuracy, across the datasets.
- **Early exits + Distillation:** early exit benefits persist when applied to compressed models, highlighting the complementary nature of these optimization techniques. More specifically, DistilBERT-EE incurs 10.5% lower compute and latency values relative to DistilBERT, with almost identical accuracy (within 0.14%).

2.3 Challenge: Batching in EE-DNNs

Despite the near-ideal characteristics for inference that EE-DNNs intuitively offer, practical usage for EE-DNNs have been limited in practice.

A fundamental requirement for achieving optimal throughput in ML training and inference is the ability to batch inputs. Batching enables accelerators like GPUs to utilize all of their constituent cores and maximize the parallelism they offer. Unfortunately, EE-DNNs are fundamentally at odds with batching. Paradoxically, inputs exiting a model early (i.e., at intermediate layers) to yield compute reductions also results in *shrunken* batch sizes for the rest of the model's inference. This, in turn, fails to saturate accelerators and leads to poor resource utilization. Figure 3 illustrates this behavior: roughly half of the samples exit halfway through the model (by ramp 6), which cuts GPU utilization by more than 25% for the remainder of the model execution. In summary, there exists a fundamental tension between compute savings and resource utilization with EE-DNNs.

One workaround to this natural tension is to make all inputs in the whole batch exit at ramps, which maintains high resource utilization and eliminates the overhead associated with reforming a batch after certain samples exit. However, as the batch size increases, the probability of all of the samples in the batch exiting at the same ramp decreases exponentially, limiting the feasibility of this approach. As a result, existing early-exit networks have restricted the use of batching [39, 49, 59, 68–70, 73], negating their benefits.

2.4 Real World Importance

Here, we describe our experience in using EEs in a large-scale production service. Although, we are unable to list in-depth operational details due to their business critical nature of the service, our aim is to highlight the practical utility of EEs in real industrial settings and materialize the challenges above as the primary impediments to such deployments.

The infrastructure runs some of the world's largest enterprise inference workloads. Of several services it supports, one particular service has been under active development to lower the computation requirements since the projected costs were prohibitive in nature. Concretely, the service performs document classification and ranking to several underlying tasks, and started with a derivative of the 12-layer BERT-BASE model. The service handles *many billion requests per day* and the projections indicate exponential increase, drawing a team-wide focus on imposing a compute cost budget per input without sacrificing SLO constraints.

While the 12-layer version delivered the best accuracy, the cost per input was prohibitive, with projections of multi-million \$ overheads atop the budget. Following this, the team resorted to off-the-shelf compression techniques, specifically a combination of knowledge distillation and pruning, that generated 6- and 3-layer variants of the model. The 6-layer version met accuracy targets, but still considerably exceeded the per-input compute cost. In contrast, the 3-layer version met that compute cost, but brought $\approx$4% accuracy loss.

The service then turned to EEs on the 12-layer version which not only satisfied the per-input compute cost (subject to the SLO), but also delivered accuracies that closely mirrored (within 1%) the original 12-layer model. Unfortunately, the lack of batching in EEs (§2.3) proved to be the showstopper, especially for a service that needs to serve many billion requests. An alternate solution aimed at developing custom hardware to support EE via a streaming mechanism was attempted but quickly abandoned due to prohibitive costs. As a result, the service compromised and resorted to the 3-layer version with accuracy loss.

Based on our experience, we posit that if the batching and thus the resource utilization problem were solved, there exist many practical use cases that would greatly benefit from EEs

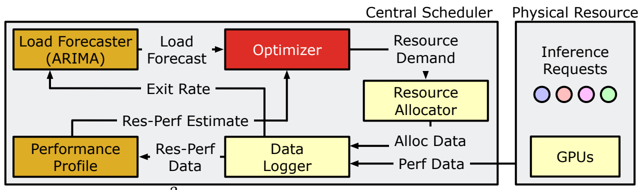

Figure 4. E^3 **architecture. Arrows indicate data flow.**

at large scale (including our own) – enabling such practical use of EEs is the core focus of this paper. While the bulk of our evaluation is on real-world motivated usecases that mimic our production scenario by using the 12-layer BERT language model, we also show that the techniques we present extend to autoregressive LLMs in §5.1.3.

3 E^3 Design

E^3 (fig. 4) seeks to mitigate the limitations of early-exit networks and utilize them to provide efficient inference. The key idea E^3 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, E^3 needs to determine the optimal number of splits, and the optimal number of GPUs to run the splits on. For the former, E^3 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 scheduler. We describe each component in detail.

Importantly, E^3 does not assume any knowledge or make any assumptions about the inner-workings of the early-exit mechanism or the model; instead, it generalizes to any EE-DNN. The only requirement for E^3 is that it is able to query the batch size at every exit ramp (for profiling). We show in §3.4 that additionally granting E^3 the freedom to *disable* an exit ramp (e.g., by using a model-provided API) can result in additional performance benefits. However, this is *not* a requirement.

3.1 Online Batch Profile Estimation

E^3 makes the determination of the optimal number of splits for an EE-DNN based on the batch size reduction characteristics. Thus, it must determine how batch size changes over the course of execution of the EE-DNN. Since inference workloads are time-varying [34], E^3's batch profiling must operate in an online fashion. To do this, E^3 uses ARIMA [37], a timeseries forecasting method.

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 (corresponding to each layer in the model), and each exit is annotated with the batch size. The model ingests inputs at a batch size

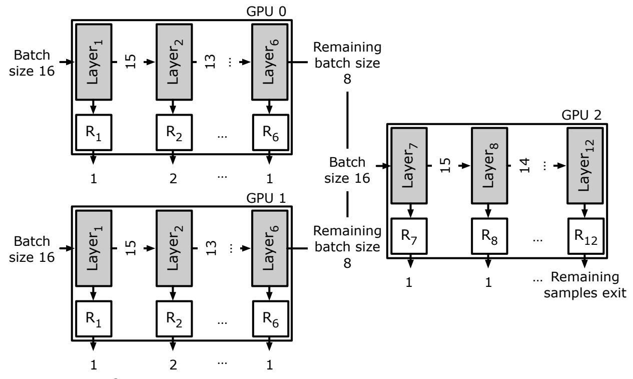

Figure 5. E^3 uses a model-parallel execution strategy to break up EE-DNNs into splits that keep batch sizes constant, and run those splits across (potentially) heterogeneous hardware. R_n indicates the exit ramp after layer n.

of 16. The estimator outputs its forecast of the expected batch sizes at each of the exit ramps in a rolling fashion. The estimation provides the optimizer with the relative decrease in batch size, e.g., the batch-size shrunk to 50% by layer 3. Due to the time-varying nature of the workload, the estimator runs continuously, and we show the performance and efficacy of E^3's online batch profile estimator in §5.

We emphasize that *perfect prediction is not required*, as the output of the estimator is just a guide in E^3's optimization (§3.2). E^3's scheduler includes safety checks to ensure that the predicted values never exceed the maximum possible batch sizes that can be supported by the resources. Mild prediction inaccuracies do not alter E^3's performance, and larger inaccuracies only affect the magnitude of E^3's gains (not correctness), e.g., predicting a lower batch size will reduce the realizable gains. Nevertheless, fig. 21 shows that E^3's batch size profiles closely match reality. For example, if we predict a batch shrinkage of 50% by layer 6, split the model and run an input batch size of 16 (fig. 5), E^3 does not expect (nor rely on) the output at layer 6 to be exactly 8. Rather E^3 only needs it to be close to 8 for the best performance (not correctness), and can handle any size as long as it can safely merge multiple of such batches at the next layer without overshooting the available resources (§5.8.2).

The use of a timer-based batch profile estimation may be problematic when unexpected spikes occur. E^3 handles this in the same way that traditional inference pipelines do: we incorporate a slack for the SLO in the optimizer (§3.2), and can use buffer resources if available. In its current design, E^3 drops requests that cannot be served, similar to Clockwork [29]. Since E^3 monitors its estimated batch profile compared to observed (fig. 4), it can reactively re-run the optimizer if they differ drastically. We defer the exploration of proactive workload spike mitigation techniques to future work.

3.2 Dynamic Programming based Optimization

The objective of E^3 is to maintain the batch size nearly constant during the execution of the EE-DNN. In our earlier example of an 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[2]. 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, E^3 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. E^3'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*:

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

where $B_{0\to N}$ is the estimated batch profile for the EE-DNN model with N layers (i.e., how the batch size shrinks from layer 0 to N). Since the request rate is R, we can estimate the largest batch size, B_0, that does not violate the SLA. Using these definitions, the throughput of the system is

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

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

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

where $Slack$ is the allowed slack in SLO ($\geqslant 0$), $Cost_{baseline}$ is the cost of the baseline model, α is a cost multiplier. We can define a dynamic programming based recursive optimization:

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

where $T(i \to j, B_{i\to j}) = \sum_{k=i}^{j} P(k, B_k)$. In this formulation, P is the throughput-latency profile (throughput and latency of layer k with batchsize B_k), 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.

[2]The splits contain equal number of layers here, but this need not be the case.

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

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

where:

P is the throughput-latency profile for GPU config c

$B_{0\to N}$ is the est. batch profile for EE-DNN with N layers

B_k is the est. batch size at layer k; each GPU handles 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 6. The optimization formulation in E^3

3.2.1 Leveraging Model Parallelism

In the previous formulation, the splits of the EE-DNN are executed in a single GPU in a serial fashion, or multiple GPUs in a data parallel fashion, where each GPU executes the splits sequentially. This may be optimal for some cases (E^3's optimizer considers all placement choices and will pick it as the choice in such scenarios), but in practice the request rate (R) is large enough to warrant the use of a GPU cluster. This provides E^3 with the opportunity to execute the splits on different GPUs in parallel (fig. 5), commonly referred to as *inter-layer model-parallelism*. Further, each split can be replicated *independently*. For a cluster with m machines, we can modify E^3's optimization as:

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

where the first split is replicated on $m-m'$ machines, $T(i \to j, m, B_{i\to j}) = \sum_{k=i}^{j} P(k, m, B_k)$ so that each GPU (machine) processing $\frac{B_k}{m}$ samples. T_x is the communication time for sending data from the end of a split to the next. In addition to minimizing the number of splits, the formulation also tries to minimize the resources to run the splits.

3.2.2 Incorporating Pipelining

Due to the use of model parallelism, E^3 may incur GPU underutilization if communication costs dominate. To mitigate this, we adopt a simple pipelining strategy. Each GPU processing a split can process the next batch once it is done with the current batch, thus allowing overlapping computation and communication. In the steady state of such a pipeline, E^3's optimization can be modified to optimize $A(i \to j, m, B_{i\to j})$:

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

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

E^3 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 E^3's model parallel execution strategy. For instance, each split can have different computational requirements, and placing the split on the right hardware configuration can both reduce cost and improve utilization. Towards this, E^3 incorporates heterogeneity in its optimization formulation by accounting for the configuration of the GPUs available (e.g., A100) within the constraint that the replicas of each split can only be placed on the same type of GPU. Figure 6 shows the final optimization formulation.

3.3 Heterogeneity Aware Model-Parallel Execution

E^3'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. It 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 [29], the scheduler knows exactly the amount of time necessary to execute each split. Using the output from the optimizer, E^3'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.

Each split independently executes batches, and upon completion of the batch, immediately moves on to the next. The machine hosting the next split maintains a queue that holds the partial results until it has received such inputs from others. A potential problem in this pipelined execution strategy arises if the execution times of splits are imbalanced—queues may build up and result in SLO misses. E^3 sidesteps this by explicitly considering execution time of splits when determining which splits to use and where to place them. Even with this, however, it is possible for some GPUs to become stragglers [33]. For this, E^3'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. Note that E^3 inherits the low-level execution decisions (e.g., w.r.t SLO violations) of the platform on which it is implemented.

3.4 Improving E^3 by Relaxing Assumptions

So far, the techniques we have outlined made no assumptions about the EE-DNN, its exit strategies, or ramps. E^3's efficacy can be further improved if this assumption is relaxed, and it is granted more control over the EE-DNN. For instance, providing information about the exit strategy can let E^3 control the exit in real-time if desirable. To do so, E^3 provides a simple wrapper function, `exit-wrapper`, that a developer of an EE-DNN would use to wrap the exit checking logic with.

E^3 can use 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, E^3's wrapper can be used to take decisions per ramp independently. For EE-DNN architectures where exits are *dependent*, i.e., the decision to exit at a ramp is made using information from earlier ramps, E^3 keeps track of this information to determine whether the logic has to be executed within a split. These two styles of ramp architectures that E^3's wrapper supports account for a large fraction of EE-DNNs. A simple use-case here is to disable ramps that are not useful. Sophisticated use-cases such as real-time ramp tuning are possible; we leave them for future work.

Regardless, we reiterate that using the wrapper is not a necessity – indeed, our evaluation results assume that the wrapper is not used (we evaluate it in §5.8.6)– it simply provides an opportunity to improve E^3's performance.

4 End-to-End Inference & Implementation

We implemented E^3 as a layer on PyTorch [8] and use TorchServe [10] to serve inference requests using a REST API. We optimize GPU serving by converting models to ONNX [7] and using TorchServe's native ORT support [5]. While this closely mimics our production scenario (§2.4), it does not depend on any platform specifics and can be ported to other inference frameworks, e.g., NVIDIA Triton [4].

The end-to-end inference pipeline supports both closed- and open-loop clients (§5). E^3 takes a EE-DNN model as input and automatically splits and replicates it using its techniques. Each instance of a split of the model maintains its own queue and executes requests in batches. For closed-loop clients, the batching is static; the scheduler simply waits for the right batch to be formed before feeding into the first split. For open-loop clients and workloads with variable request rates (e.g., Twitter trace in fig. 19), just like other serving systems, E^3 follows dynamic batching by queuing incoming requests and waiting until it either has the target batch size or the queued inputs would violate SLAs if not immediately scheduled; on either criteria, E^3 dispatches the corresponding inputs. The scheduler incorporates a slack for the SLO (20% in our evaluation), and requests that cannot be served are dropped (§3.1). By default, E^3 runs its splitting optimization every 2 minutes; this frequency outpaces the variability observed in our production environment (on the order of hours), but we chose 2 minutes to highlight the lightweight nature of the process (fig. 20). E^3 leverages state-of-the-art serving platforms for its reconfiguration needs. These platforms support transparent scaling in both directions (e.g., during load variations) that E^3 hooks into when a change to the splits is necessary.

With open-loop clients and varying request rates, EE-DNNs in general pose a challenge in terms of estimating the execution time necessary for dynamic batching due to the adaptive nature of execution. However, because E^3 solves the batching problem by reducing the model into independent, replicated pieces that are (relatively) stable in exit rate (and thus, input/output batch sizes and processing time), it is able to resolve the tension between batch decisions and variability from exit rates, making it similar to the closed-loop setting.

5 Evaluation

We evaluate E^3 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, E^3 is able to provide up to 1.70× and 1.74× better goodput compared to SOTA EE-DNN models in NLP and computer vision, respectively. E^3 also outperforms stock DNN models by up to 1.32× and 1.58×. Additionally, E^3's improvement increases with increase in batching opportunities or availability of heterogeneous resources (§5.1).
- E^3's win generalize to autoregressive LLMs and can further complement compression. E^3 boosts the performance of compressed models by up to 1.67×. On LLMs, E^3 achieves 2.84× and 3.8× better goodput compared to stock models, in translation and summarization tasks, respectively. (§5.1.2, §5.1.3)
- When the performance requirements, such as the desired throughput, are fixed, E^3 is able to achieve them at substantially lower cost: E^3 incurs 35% to 78% lower cost depending on the batching opportunities (§5.3).

Experimental Setup: We run experiments with 4 different NVIDIA GPUs – A6000, V100, P100, K80, and consider a cluster with 46 GPUs spread across 26 machines. 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 connected via a shared PCIe interconnect, and servers are interconnected via 10 Gbps Ethernet. While this setting test E^3 under constraints, we note that faster interconnects (e.g., 40/100 Gbps links, NVLink) would benefit E^3 and further improve its performance. All servers run 64-bit Ubuntu 16.04 with CUDA library v10.2 and PyTorch v1.6.0.
Datasets and models: We use numerous state-of-the-art models to evaluate E^3. For non-generative vision tasks, we primarily use ResNet-50 [35] from TorchVision [51] and BERT-BASE, LARGE from Transformers [11] for NLP tasks. For autoregressive tasks, we use T5 [58] and Llama [66], and for compressed model we use DistilBERT [57]. Following previous work [29, 56], we run ImageNet [22] and the GLUE [2] benchmark for non-generative tasks, and WMT [16] and Samsum [26] for autoregressive tasks in closed-loop clients. We evaluate open-loop clients in §5.7.

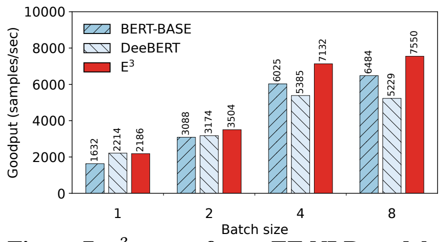

Figure 7. E^3 **outperforms EE NLP models by upto 1.44× in homogeneous settings.**

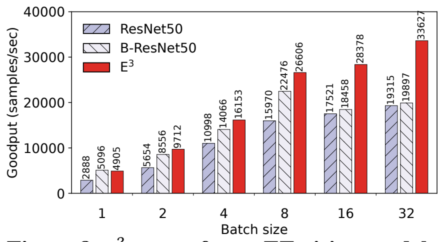

Figure 8. E^3 **outperforms EE vision models by upto 1.74× in homogeneous settings.**

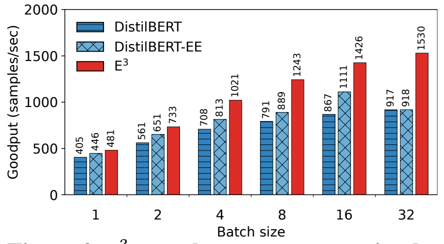

Figure 9. E^3 **complements compression by augmenting its performance by up to 1.67×.**

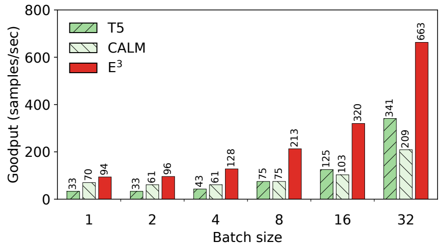

Figure 10. When applied to translation tasks, E^3 **can improve performance of stock LLMs by up to 2.84×.**

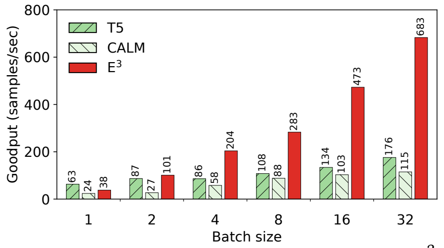

Figure 11. In summarization tasks, E^3 **brings up to 3.8× improvements in LLMs (average output length: 18 tokens).**

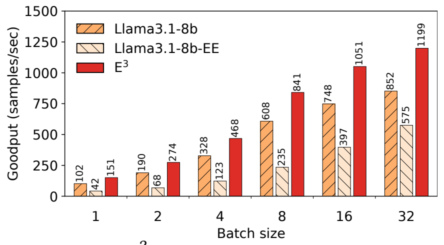

Figure 12. E^3**'s techniques extend to the decoder only LLMs. On Llama, the gains are up to 1.48×.**

Comparison & Metrics: We primarily evaluate E^3 against BranchyNet [64], DeeBERT [69] and CALM [58], three representative EE-DNNs in the vision, non-generative and autoregressive NLP domains respectively (§5.6 considers other EE architectures). Since we are unaware of an early-exit model for compressed models, we develop one for DistilBERT to show the complementary nature of E^3 (and EEs in general) to compression (§5.1.2). Results in deep-dive experiments focus on the NLP model, but we note that the shown trends persist for all considered models. 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 (§5.8.4 evaluates other values), which results in less than 2% error, an acceptable value in our production scenarios.

Our main metric of comparison is *goodput*, or the number of samples per second that can be sustained without violating SLOs. We use a default SLO of 100 ms (in line with prior work [29, 61]), but consider other values in §5.8.5. We use batching by default and consider different batch sizes; results are shown for all batch sizes that avoid SLO violations. Reported numbers include queuing delays (if applicable) and model-parallel overheads for E^3, unless specified. We note that the workloads considered mimic the high level characteristics (e.g., average arrival rate, SLO) of the traces we observe in production, and that our results are consistent with those we've seen when trying this approach in practice.

Workloads: We use two kinds of workloads in the paper. Following previous work [29, 30, 56] and many subsequent works, we use the arrival rates in the open source Twitter trace scaled up to simulate an open source variable rate trace (due to the extreme bursty nature of this trace, we couldn't scale it beyond an average of 1000 req/s). Second, we use both uniform arrivals and our own production-based hyper parameters to simulate workloads to E^3, scaled down to our available hardware resources. We note that due to the business critical nature of the enterprise workloads, the company does not allow non-business use of raw requests (e.g., we couldn't access the request to replay it through E^3), and due to the volume of requests, the infrastructure doesn't log metadata on each request-instead it only records statistics on load periodically. Thus, we use the open source GLUE inputs scaled to mimic our setting. The production infrastructure typically processes several million requests per second; which when scaled to our resources averages 9,000 requests per second for NLP with a variance of approximately 5%. To examine hardness, we analyze the GLUE dataset to bin them to easy and hard inputs; and then varied their ratio. Our real-world real-world scenarios predominantly exhibit the 80% easy, 20% hard input mix which is a favorable scenario for E^3.

5.1 Goodput Improvements

We first show the goodput obtained by E^3 and its comparison systems for various batch sizes, assuming the cost to be constant, i.e., both E^3 and comparison systems use resources that cost the same.

5.1.1 Non-Generative Models In figs. 7 and 8, we depict the performance of E^3 when the cluster is made up of *homogeneous* resources, specifically 16 V100 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, EE-DNN model becomes progressively worse compared to the non-EE model, BERT-BASE, which is now able to utilize the parallelism offered by the GPU. E^3 on the other

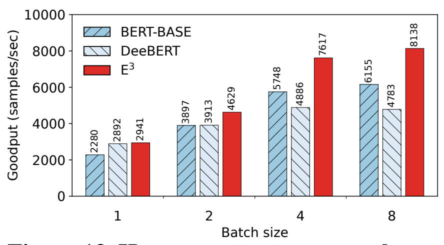

Figure 13. Heterogeneous resources boost E^3's NLP performance to up to 1.7×.

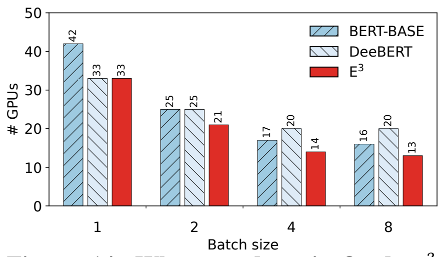

Figure 14. When goodput is fixed, E^3 achieves it using less resources.

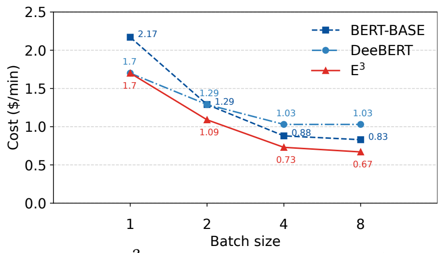

Figure 15. E^3 incurs the lowest cost (up to 35% lower) for a fixed goodput.

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, E^3 incurs a small penalty due to its model-parallel execution. E^3's performance improvement increases with increase in batch size, and provides up to 1.44× increase in goodput compared to DeeBERT, and up to 1.30× compared to BERT-BASE. Note that while the models do not saturate the GPU at a batch size of 8, the next batch size violates the SLO. The improvements are bigger in vision models, where E^3 is able to provide up to 1.74× better goodput.

5.1.2 Compressed Models Next, we show that E^3 is complementary to compression. Since there are no available early-exit variants of compressed models, we developed one in house as described in §2. We repeat the experiment above for the NLP task but replacing BERT with its compressed variant. We compare against DistilBERT-EE, our in house developed EE variant and E^3 which applies the techniques we propose in this paper on DistilBERT-EE. The results are shown in fig. 9 which shows the relative performance of E^3 compared to the compressed model.

We notice that unlike its non-compressed counterpart, compressed models are able to leverage early-exits better and that the performance gains carry over for batch sizes greater than 1. This is because a major fraction of the inputs exit right at the middle of the model (after layer 3). However, as the batch size increases, the relative performance of the EE variant degrades. In contrast, E^3 is able to provide significant benefits, achieving up to 1.67 × improvements in goodput.

5.1.3 Auto-Regressive Large Language Models (LLMs) Now we evaluate whether our techniques generalize to emerging autoregressive, large language models (LLMs). We consider the recent CALM architecture [58] that enables EE on T5 LLM. We evaluate E^3 on two tasks: machine translation using the WMT dataset [16] and document summarization using the samsum dataset [26]. Using CALM paper's default configuration (softmax confidence measure with a threshold of 0.25), we find that approximately 70% of the inputs exit by layer 2 (of 8 decoder layers). E^3 thus splits the model into two parts at the end of layer 2. Due to the compute and memory requirements of the T5 model, we use 4 NVIDIA A6000 GPUs for this experiment.

Figure 10 compares the goodput with E^3, T5, and CALM for the translation task, while fig. 11 compares the goodput when applied to the summarization task. As shown in fig. 10 and in line with Table 2 of the CALM paper, CALM brings 2.84× goodput increases compared to T5 for batch size of 1 (CALM does not support batching as discussed in Appendix C of [58]). However, benefits quickly diminish as batch sizes grow. In contrast, E^3 maintains its speedup for all batch sizes. E^3's benefits are even bigger when applied to the summarization task, which generates variable length outputs (we observed an average length of 18 tokens per output in this experiment). Here, we notice an improvement of up to 3.8×.

We do not use CALM as a primary baseline in other evaluations due resource constraints and because CALM's exit methodology assumes an encoder-decoder architecture to enable exiting (e.g., T5's encoder state) that may not carry over to decoder-only architectures (e.g., GPT family). Iterative scheduling/continuous batching is a technique introduced in Orca [72] to resolve LLM batching inefficiency *across* iterations. However, each iteration in an LLM consists of computing over the entire model and thus the EE-batch shrinking problem remains *within* an iteration (our focus). Since CALM (decoder-only LLMs) currently cannot support iterative scheduling (early-exiting), we defer synergizing iterative scheduling with E^3 to a future work.

Llama family: We further investigate whether E^3's benefits carry over to decoder-only LLMs. Due to the lack of early-exit variants of such models, we resort to the guidance in recent work [14], we selected the 8 billion variant of Llama 3.1 [3] and replicate the final layer as the exit ramp. Unfortunately, this alone is insufficient to convert the model into its early-exit variant, as the problem of managing the KV cache still needs to be resolved. To the best of our knowledge, this is an open problem: while CALM proposes a solution to this problem, it is focused on encoder-decoder architecture and not on decoder only models such as Llama. Therefore, we restrict ourselves to the Google BoolQ question answering dataset [19] where the output is a single token (yes/no). We note that this restricts the gains E^3 can possibly obtain, as, unlike in the previous case with CALM, there are no multiple iterations to benefit from. Nevertheless, this experiment still shows the generalizability of E^3 to a different family of LLMs.

Using E^3's profiler on the BoolQ dataset using the softmax confidence showed that approximately 50% of the inputs exit after layer 25 of the Llama3.1-8b model. Thus, E^3 splits the models into two parts. We then run the experiment using different input batch sizes and show the results in fig. 12. Noteworthy is the significant difference between the vanilla and the early-exit variant of the model (Llama and Llama-EE), even for small batch sizes; even with a batch size of 1, the EE variant significantly underperforms the vanilla variant. The reason is the overhead of exit checking, due to the large vocabulary sizes in Llama3.1-8b, the cumulative overhead of checking at every layer adds up. In contrast, E^3 only needs to check for exits at the end of splits, further adding to its gains. Here, E^3 is able to outperform even the vanilla variants by up to 1.48×. We emphasize that the gains are restricted due to the limitations of the benchmark as we described earlier.

5.2 Heterogeneity in Compute Resources

Figure 13 shows the performance of E^3 when the cluster consists of *heterogeneous* resources. Here, we use a mix of V100, P100, and K80 GPUs. Since we maintain the cost to be constant, we picked two configurations of machines that maximizes the goodput: 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 notice that 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, E^3 is able to effectively utilize the different GPUs and outperform the comparisons. For each batch size, E^3 identified the optimal configuration that maximizes the goodput, providing up to 1.70× improvements.

5.3 Cost Effectiveness

In this experiment, we evaluate the ability of E^3 to reduce the cost of inference when the throughput is fixed. We fix the desired throughput to be 6000 samples per second. We then consider two settings: in a homogeneous cluster which consists of V100 GPUs, we determine the number of GPUs that are necessary to sustain the desired performance; and in a heterogeneous cluster 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. 14 and 15 respectively.

E^3 provides the best performance in all settings and all batch sizes. For small batches, no model is able to utilize the GPUs efficiently, resulting in the need to use more GPUs. As batching opportunities increase, both BERT and E^3 are able to utilize the resources better, and hence the number of resources required reduces. DeeBERT is also able to use batching sparingly, but due to the batching limitations of EE-DNNs, the resource utilization is not as efficient as E^3 or BERT, resulting in it using more resources than either of them. E^3 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, E^3 is able to provide better performance per dollar compared to its comparisons: it achieves the same performance at 35–78% lower cost.

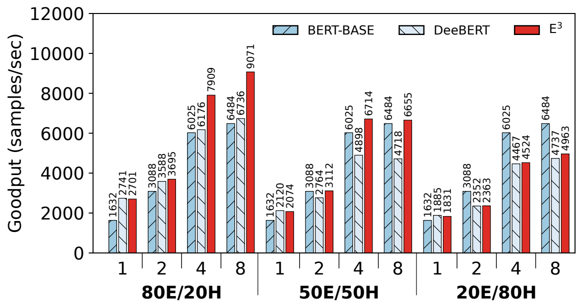

Figure 16. E^3**'s online batch profiling (§3.1) and optimizer (§3.2) are able to adapt to workload variations.**

5.4 Workload Adaptability

We now seek to answer *"Can E^3 adapt to workload variations?"* by evaluating its batch profiler and optimizer. We created three variations of the workloads by changing the ratio of the easy and hard examples to 80:20, 50:50 and 20:80. We then ran inference in closed loop on each of the models on both homogeneous and heterogeneous resources, switching between the workloads at fixed intervals. We start with the 80:20 mix, and after a specific time, we switch to the 50:50, and then to the 20:80 mix. Figure 16 shows the results.

We notice that the EE-DNNs provide benefits compared to their non EE counterparts when the batch size is small and the mix has more 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-EE models perform poorly when the workload consists of a large fraction of easy inputs, but are able to provide good performance when batch sizes are large or inputs are hard, requiring the entire predictive power of the model.

In contrast, E^3 is able to effectively adapt to the requirements of the workload. When the workload is skewed towards easy inputs, E^3 behaves like an 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. The same adaptation applies when the workload becomes mostly hard. Thus, E^3 behaves similar to the non early-exit model in this case, with the added

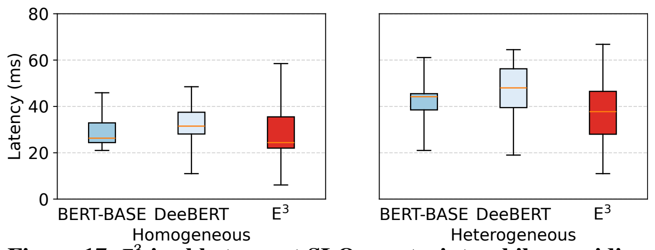

Figure 17. E^3 is able to meet SLO constraints while providing lower inference latencies at quartiles.

benefit of being able to leverage heterogeneity through its model-parallelism based execution.

5.5 Latency Implications

Since E^3 depends on a split-execution model (where parts of a model may be executed on different GPUs), and EE-DNNs impose exit-checking overheads, it is natural to assume that the benefits of E^3 come at the cost of increased latency. To evaluate the implications, we measure the latencies incurred by E^3, BERT and DeeBERT over 100K inferences. Figure 17 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 to meet the SLO.

E^3 attains the lowest min, median, 25th-%ile and 75th-%ile latencies across the board, which may seem counter-intuitive. While E^3 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 E^3, which affects the tail (max) latency. Even then, the SLO is not violated, as E^3'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).

5.6 Generality to EE Architecture

Here, we show that E^3's techniques are general and can be applied to different EE-DNN architectures. For this, we apply E^3 to PABEE [73], an EE-DNN model based on BERT that uses a sophisticated counter based mechanism to decide on the exit choice. This model represents a different architecture compared to DeeBERT. We use the setup from §5.1; fig. 18 shows that E^3 is able to provide upto 1.55× higher goodput compared to PABEE.

5.7 Extremely Bursty Workloads / Open-loop Client

Until now, we have assumed arrival patterns that mimic our production setting (i.e., there are enough requests to warrant batching continuously) that we believe emulates a real-world setting. Here, we evaluate E^3's performance in an extremely bursty scenario. Due to the lack of open-source inference workloads, we use the request arrival rate of new tweets in the Twitter trace [1] used in previous work [30, 56], scaled to have an average request rate of 1000 req/sec. Figure 19 shows that E^3 is able to maintain its performance even when the arrivals are very bursty. Further, E^3 attains 29% improvement in goodput over DeeBERT, and 16% improvement over BERT-BASE. While the improvements over non-EE model may seem low, we note that the Twitter trace has extreme bursts and long periods of inactivity (amplified when scaling to high average request rates) due to which the GPU utilization remains under 50%, resulting in little batching opportunities.

5.8 Microbenchmarks

5.8.1 Overheads 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 (GPUs and the number of layers in the EE-DNN) change. fig. 20 shows that the optimizer is lightweight.

5.8.2 Efficacy of E^3's Batch Profile Estimation E^3 depends on its online batch estimation (§3.1) to determine the model splits. We evaluated this technique as follows. 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 21 shows the predicted and the actual batch sizes on the two cuts for 10 such windows. We can observe from the results that E^3's prediction closely matches reality.

5.8.3 Batch Profile Estimation Sensitivity Analysis Although our batch profile estimation works in practice, we now evaluate how errors in prediction can affect E^3. In particular, we evaluate the gains lost due to incorrect predictions, since the correctness of E^3's execution is not affected by prediction errors (§3.1). For this evaluation, we deliberately introduce errors in the prediction as follows. We consider the Llama3.1-8b model setup in §5.1.3. For different input batch sizes, we change the prediction to include errors ranging from 0% (perfect prediction) to 100%. For example, if the batch profiler estimates an actual batch shrinkage of 50%, the expected batch size at the end of the first split is 8 with 0% error, and 12 with 50% error for an input batch size of 16. We depict the results in fig. 22. We note a slight decrease in the goodput obtained by E^3 when the prediction errors are within reasonable ranges. For an error of 20%, the goodput loss is approximately 4-8%. As expected, larger errors in prediction leads to increased loss in goodput, with substantial errors leading to worse performance for E^3 compared to vanilla models. However, significant errors can be detected easily and serve as a signal for triggering E^3's optimizer to correct them.

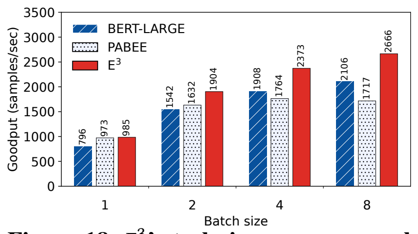
Figure 18. E^3's techniques are general and can apply to many EE-DNN architectures.

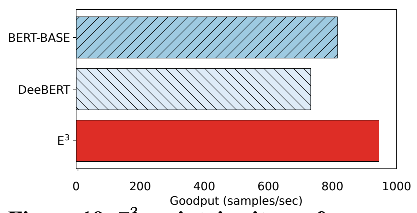
Figure 19. E^3 maintains its performance when requests are extremely bursty (Here, GPU utilization is less than 50%).

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

Figure 20. E^3's optimizer is lightweight and incurs low overheads to find the optimal split and resources needed for a EE-DNN.

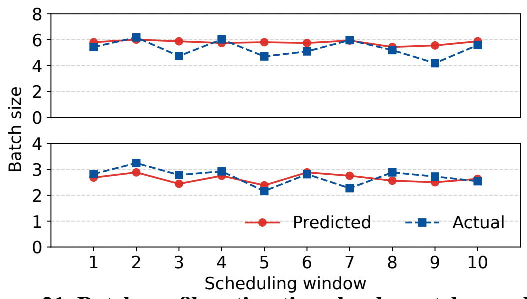
Figure 21. Batch profile estimation closely matches reality.

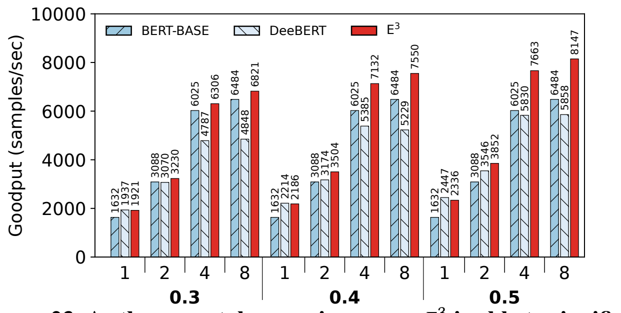
Figure 23. As the error tolerance increases, E^3 is able to significantly improve its (already good) performance.

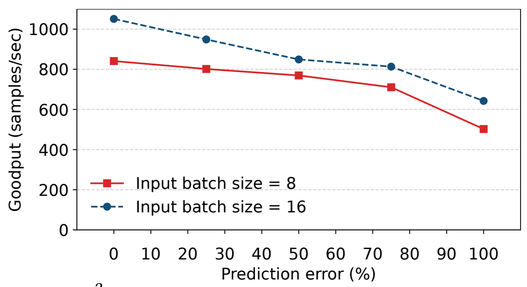
Figure 22. E^3's gains lost due to misprediction is minimal.

5.8.4 Impact of Error Tolerance We investigate how the error tolerance affects E^3'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. 23. 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, E^3 is able to identify this, and tune the splits accordingly. If the user is willing to afford more errors, E^3 is able to provide better goodputs, up to 43% higher compared to DeeBERT.

5.8.5 Impact of SLO SLOs determine the max batch size that can be created; a strict SLO translates to fewer batching possibilities and hence smaller batches, and vice-versa. We consider SLOs from 25-1000ms and translate them to the max batch sizes that can be supported. For each SLO and max batch size, we evaluate E^3 and its comparisons in fig. 24. When the SLO is small, batching opportunities are virtually nil. At small batch sizes, DeeBERT (and EE-DNNs in general) offer compelling advantage over BERT. E^3's optimizer is able to adapt; at a batch size of 1, E^3's goodput is just 1% lower compared to DeeBERT. However, as batching opportunities arise, both E^3 and BERT are able to leverage the parallelism offered by the GPU. Here, E^3 provides up to 63% (34%) higher goodput compared to DeeBERT (BERT).

5.8.6 Relaxing E^3's Assumptions Here we evaluate the usefulness of E^3's if it is able to control the EE-DNN following the simple use-case described in §3.4. In this experiment, we assume that E^3 is able to disable the exits within a split (except the last one which is required) which are not useful. The results in fig. 25 show that E^3 is able improve its performance by up to 16% by avoiding exit-checking overheads.

5.8.7 Impact of Model Parallelism With model parallelism turned off, E^3 must execute the splits in the same GPU serially, waiting for all copies of a split to finish before it can start executing the next split. Figure 26 shows that the ability to execute the splits across multiple GPUs significantly improves E^3's performance.

5.9 Shortcomings

There are two shortcomings of E^3. First, E^3 is designed for workloads where there are enough opportunities to batch the input. When this opportunity ceases to exist, E^3 does not provide benefits. Figures 7 and 8 show that for batch size 1, E^3 is up to 3% worse compared to the EE-DNN model. Our production experience indicates that small batches are rare in the real-world, hence we believe E^3 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, E^3 is unable to find optimal splits or batching opportunities for its model

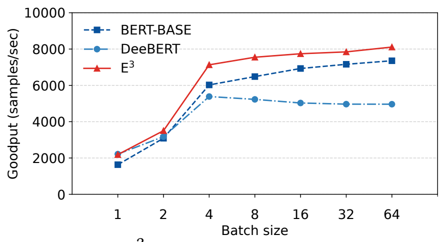

Figure 24. E^3's performance improves as batching opportunity increases.

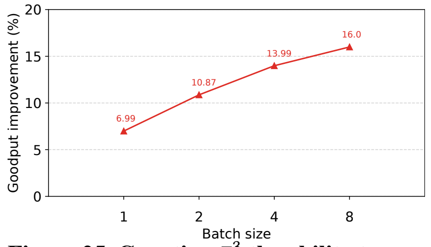

Figure 25. Granting E^3 the ability to control the EE-DNN boosts its performance.

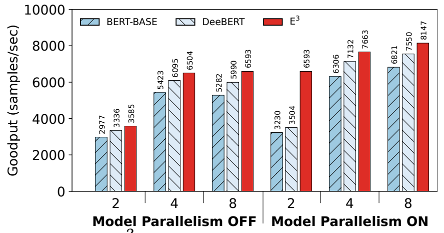

Figure 26. E^3 is able to parallelize the execution of EE-DNN model splits across GPUs.

parallel model. Figure 16 shows that E^3 is up to 23% worse compared to the non-EE model when the workload is 80% hard. Compressed models are not likely to be useful in such cases either, and would incur significant accuracy loss.

6 Additional Related Work

Deep learning systems typically focus on improving the performance of deep learning *training* [40, 53, 55]. In addition to the data parallelism and model parallelism provided by popular open-source frameworks such as PyTorch [8] and TensorFlow [9], recent works have proposed hybrid parallelism strategies. Further, pipelining and compression has been shown to boost training performance. 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 [20, 29, 30, 56, 61]. However, to the best of our knowledge, none of the existing works on inference systems focus on leveraging early-exit networks.More recently, researchers have proposed ways to mitigate the inefficiencies associated with the iterative style processing in autoregressive models [13, 45, 72]. These are orthogonal to our work, since they focus on optimizations between iterations, while E^3's focus is on optimizations within an iteration.

Several **early exit networks** (§2.2) have been proposed to accelerate inference of vision models and more recently for language models (e.g., BERT and similar multi-layer transformer models) by leveraging varying input sample complexity. Key question in EE-DNNs 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, the sample performs early exiting. Previous studies have proposed various *heuristic* criteria to judge *certainty level*. *Confidence-based criterion* [14, 49, 59, 68] interpret the label scores output by softmax as confidence scores. *Entropy-based criterion* [69, 70] rely on the entropy of predicted probability distribution to be smaller than pre-defined threshold. *Counter-based criterion* [73] require off-ramps classifiers to continuously generate identical predictions for pre-defined times. *Voting-based criterion* [63], inspired by the ensemble technique, requires a pre-defined number of off-ramp classifiers to reach an agreement. *Model-based criterion* [15] uses additional lightweight neural networks to predict the exiting decisions. [32] discusses how to train ramps in EE-DNNs. Additionally, several approaches choose not to rely on heuristic criteria, and introduce an additional module which *learns-to-exit*. Often it is a simple one-layer fully-connected network, which is shared among all off-ramps and outputs the certainty level [47, 74]. As we show in this paper, EE-DNNs suffer from fundamental challenges (§2). E^3 overcomes these to make EE-DNNs practical for inference.

Mixture-of-Experts (MoE) are **dynamic neural networks** that support input adaptation like EE-DNNs, but by routing inputs to different sub-networks [25, 31, 38, 48, 60, 71]. We believe that E^3 is complementary, e.g., each expert can employ EE techniques. BE3R [50] poses EEs as a MoE problem where each expert is the model replica with an increasing number of layers and routing sub-batches of inputs to the right replica. This avoids the batching challenge but relies on routing correctness, sufficiently large sub-batches per expert, and more resources. Brainstorm [21] optimizes the *execution* of dynamic neural networks in the GPU. Such optimizations can complement E^3 and boost its performance further. Tabi [67] runs inputs through a smaller model, only invokes a larger model for low-confidence inputs, and suffers from similar batching issues that can be addressed with techniques in E^3.

7 Conclusion

E^3 addresses the detrimental relationship between compute savings (from exits) and resource utilization (from batching) that EE-DNNs fundamentally bring. The main idea behind E^3 is to split and replicate layer blocks to keep batch sizes constant throughout execution and efficiently take advantage of diverse resources. We find that E^3 can deliver up to 1.74× improvement in goodput (for a fixed cost) or 1.78× reduction in cost (for a fixed goodput). Further, E^3's wins extend to autoregressive LLMs (up to 3.8×) and can complement compression by boosting its goodput by up to 1.67×.

Acknowledgements. We thank the SOSP reviewers, and our shepherd for their help in improving this work. During the initial phases of this work, Anand Iyer was a researcher and Swapnil Gandhi was an intern at Microsoft Research.

References

[1] [n. d.]. ArchiveTeam JSON Download of Twitter Stream 2018-04. https://archive.org/details/archiveteam-twitter-stream-2018-04/.

[2] [n. d.]. GLUE Benchmark. https://gluebenchmark.com/.

[3] [n. d.]. Llama Model Family. https://www.llama.com/.

[4] [n. d.]. NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

[5] [n. d.]. ONNX Run Time. https://github.com/microsoft/onnxruntime.

[6] [n. d.]. ONNX Runtime serves over 1 trillion daily inferences at Microsoft. https://news.microsoft.com/source/features/ai/how-microsofts-bet-on-azure-unlocked-an-ai-revolution/.

[7] [n. d.]. Open Neural Network Exchange (ONNX). https://onnx.ai/.

[8] [n. d.]. PyTorch. https://pytorch.org/.

[9] [n. d.]. TensorFlow. https://www.tensorflow.org/.

[10] [n. d.]. TorchServe. https://pytorch.org/serve/.

[11] [n. d.]. Transformers. https://github.com/huggingface/transformers.

[12] 2021. Live Video Analytics with Microsoft Rocket for reducing edge compute costs. https://techcommunity.microsoft.com/t5/internet-of-things/live-video-analytics-with-microsoft-rocket-for-reducing-edge/ba-p/1522305

[13] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[14] Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. 2023. Fast and Robust Early-Exiting Framework for Autoregressive Language Models with Synchronized Parallel Decoding. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5910–5924. https://doi.org/10.18653/v1/2023.emnlp-main.362

[15] Arjun Balasubramanian, Adarsh Kumar, Yuhan Liu, Han Cao, Shivaram Venkataraman, and Aditya Akella. 2021. Accelerating deep learning inference via learned caches. *arXiv preprint arXiv:2101.07344* (2021).

[16] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Barry Haddow, Matthias Huck, Chris Hokamp, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Carolina Scarton, Lucia Specia, and Marco Turchi. 2015. Findings of the 2015 Workshop on Statistical Machine Translation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Lisbon, Portugal, 1–46. https://doi.org/10.18653/v1/W15-3001

[17] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[18] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. https://doi.org/10.48550/ARXIV.1710.09282

[19] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions. In *NAACL*.

[20] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[21] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, Lidong Zhou, Quan Chen, Haisheng Tan, and Minyi Guo. 2023. Optimizing Dynamic Neural Networks with Brainstorm. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 797–815. https://www.usenix.org/conference/osdi23/presentation/cui

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, Article arXiv:1810.04805 (Oct. 2018), arXiv:1810.04805 pages. arXiv:1810.04805 [cs.CL]

[24] Angela Fan, Edouard Grave, and Armand Joulin. 2020. Reducing Transformer Depth on Demand with Structured Dropout. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SylO2yStDr

[25] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research* 23, 1 (2022), 5232–5270.

[26] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Association for Computational Linguistics. https://doi.org/10.18653/v1/d19-5409

[27] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. 2020. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*. Association for Computational Linguistics, Online, 143–155. https://doi.org/10.18653/v1/2020.repl4nlp-1.18

[28] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

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

[31] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 11 (2021), 7436–7456.

[32] Yizeng Han, Yifan Pu, Zihang Lai, Chaofei Wang, Shiji Song, Junfeng Cao, Wenhui Huang, Chao Deng, and Gao Huang. 2022. Learning to Weight Samples for Dynamic Early-Exiting Networks. In *European Conference on Computer Vision*. Springer, 362–378.

[33] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 418–430. https://proceedings.mlsys.org/paper/2019/file/84d9ee44e457ddef7f2c4f25dc8fa865-Paper.pdf

[34] 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. 2018. Applied

Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[36] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. *arXiv e-prints*, Article arXiv:1503.02531 (March 2015), arXiv:1503.02531 pages. arXiv:1503.02531 [stat.ML]

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

[38] Weizhe Hua, Yuan Zhou, Christopher M De Sa, Zhiru Zhang, and G Edward Suh. 2019. Channel gating neural networks. *Advances in Neural Information Processing Systems* 32 (2019).

[39] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. arXiv:1703.09844 [cs.LG]

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

[41] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[42] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research* (2018).

[43] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019. TinyBERT: Distilling BERT for Natural Language Understanding. *arXiv e-prints*, Article arXiv:1909.10351 (Sept. 2019), arXiv:1909.10351 pages. arXiv:1909.10351 [cs.CL]

[44] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[45] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[46] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*. PMLR, 5958–5968.

[47] Kaiyuan Liao, Yi Zhang, Xuancheng Ren, Qi Su, Xu Sun, and Bin He. 2021. 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*. Association for Computational Linguistics, Online, 2013–2023. https://doi.org/10.18653/v1/2021.naacl-main.162

[48] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime neural pruning. *Advances in neural information processing systems* 30 (2017).

[49] Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. 2020. FastBERT: a Self-distilling BERT with Adaptive Inference Time. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 6035–6044. https://doi.org/10.18653/v1/2020.acl-main.537

[50] Sourab Mangrulkar, Ankith MS, and Vivek Sembium. 2022. BE3R: BERT based Early-Exit Using Expert Routing. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3504–3512.

[51] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[52] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. 2020. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 5191–5198.

[53] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[54] Jon Porter. 2023. ChatGPT continues to be one of the fastest-growing services ever. https://www.theverge.com/2023/11/6/23948386/chatgpt-active-user-count-openai-developer-conference.

[55] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[56] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[57] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[58] Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. *Advances in Neural Information Processing Systems* 35 (2022), 17456–17472.

[59] Roy Schwartz, Gabriel Stanovsky, Swabha Swayamdipta, Jesse Dodge, and Noah A. Smith. 2020. The Right Tool for the Job: Matching Model and Instance Complexities. In *Proceedings of the 58th Annual*

Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Online, 6640–6651. https://doi.org/10.18653/v1/2020.acl-main.593

[60] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[61] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[62] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2020. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 05 (Apr. 2020), 8815–8821. https://doi.org/10.1609/aaai.v34i05.6409

[63] Tianxiang Sun, Yunhua Zhou, Xiangyang Liu, Xinyu Zhang, Hao Jiang, Zhao Cao, Xuanjing Huang, and Xipeng Qiu. 2021. Early Exiting with Ensemble Internal Classifiers. *CoRR* abs/2105.13792 (2021). arXiv:2105.13792 https://arxiv.org/abs/2105.13792

[64] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. *arXiv e-prints*, Article arXiv:1709.01686 (Sept. 2017), arXiv:1709.01686 pages. arXiv:1709.01686 [cs.NE]

[65] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2022. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. https://doi.org/10.48550/ARXIV.2204.12013

[66] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[67] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.

[68] Keli Xie, Siyuan Lu, Meiqi Wang, and Zhongfeng Wang. 2021. Elbert: Fast Albert with Confidence-Window Based Early Exit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 7713–7717. https://doi.org/10.1109/ICASSP39728.2021.9414572

[69] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 2246–2251. https://doi.org/10.18653/v1/2020.acl-main.204

[70] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. 2021. 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*. Association for Computational Linguistics, Online, 91–104. https://doi.org/10.18653/v1/2021.eacl-main.8

[71] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. 2019. Condconv: Conditionally parameterized convolutions for efficient inference. *Advances in Neural Information Processing Systems* 32 (2019).

[72] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[73] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. 2020. BERT Loses Patience: Fast and Robust Inference with Early Exit. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 18330–18341. https://proceedings.neurips.cc/paper/2020/file/d4dd111a4fd973394238aca5c05bebe3-Paper.pdf

[74] Wei Zhu. 2021. 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)*. Association for Computational Linguistics, Online, 2968–2980. https://doi.org/10.18653/v1/2021.acl-long.231