Apparate: Rethinking Early Exits to Tame Latency-Throughput Tensions in ML Serving

Yinwei Dai[*§] Rui Pan[*§] Anand Iyer[†] Kai Li[§] Ravi Netravali[§]
[§]Princeton University [†]Georgia Institute of Technology

Abstract

Machine learning (ML) inference platforms are tasked with balancing two competing goals: ensuring high throughput given many requests, and delivering low-latency responses to support interactive applications. Unfortunately, existing platform knobs (e.g., batch sizes) fail to ease this fundamental tension, and instead only enable users to harshly trade off one property for the other. This paper explores an alternate strategy to taming throughput-latency tradeoffs by changing the granularity at which inference is performed. We present Apparate, a system that automatically applies and manages early exits (EEs) in ML models, whereby certain inputs can exit with results at intermediate layers. To cope with the time-varying overhead and accuracy challenges that EEs bring, Apparate repurposes exits to provide continual feedback that powers several novel runtime monitoring and adaptation strategies. Apparate lowers median response latencies by 40.5–91.5% and 10.0–24.2% for diverse CV and NLP classification workloads, and median time-per-token latencies by 22.6–77.9% for generative scenarios, without affecting throughputs or violating tight accuracy constraints.

1 Introduction

Machine Learning (ML) inference has become a staple for request handling in interactive applications such as traffic analytics, chatbots, and web services [33, 52, 54, 74]. To manage these ever-popular workloads, applications typically employ serving platforms [7, 8, 20, 30, 44, 51, 58, 80] that ingest requests and schedule inference tasks with pre-trained models across large clusters of compute resources (typically GPUs). The overarching goals of serving platforms are to deliver sufficiently *high throughput* to cope with large request volumes – upwards of billions of requests per day [4, 50] – while respecting the service level objectives (SLOs) that applications specify for response times (often 10–100s of ms).

Unfortunately, in balancing these goals, serving platforms face a challenging tradeoff (§2.1): requests must be batched for high resource efficiency (and thus throughput), but larger batch sizes inflate queuing delays (and thus per-request latencies). Existing platforms navigate this *latency-throughput tension* by factoring only tail latencies into batching decisions and selecting max batch sizes that avoid SLO violations. Yet, this trivializes the latency sensitivity of many interactive applications whose metrics of interest (e.g., user retention [28, 70], safety in autonomous vehicles [68]) are also influenced by how far below SLOs their response times fall.

This paper explores the role that early exits (EEs) – an adaptation mechanism that has garnered substantial ML research interest in recent years [35, 48, 69, 75–77, 84] – can play in resolving this tension. With EEs, intermediate model layers are augmented with ramps of computation that aim to predict final model responses. Ramp predictions with sufficiently high confidence (subject to a threshold) exit the model, foregoing downstream layers and bringing corresponding savings in both compute and latency. The intuition is that models are often overparameterized (especially with recent growth [40, 64]), and 'easy' inputs may not require complete model processing for accurate results. Importantly, unlike existing platform knobs (e.g., batch size) that simply walk the steep latency-throughput tradeoff curve, EEs rethink the granularity of inference on a per-input basis. This provides a path towards lowering request latencies without harming platform throughputs. Indeed, across CV and NLP workloads, we find that optimal use of EEs brings 24.8–94.0% improvement in median latencies for the same accuracy and throughput.

Despite the potential benefits, EEs are plagued with practical challenges that have limited their impact (§2.3). The primary issue is that EE proposals have solely come in the context of specific model architectures that impose fixed ramp designs and locations [14, 60, 69, 76]. The lack of guidance for integrating EEs into arbitrary models is limiting, especially given the ever-growing model offerings in the marketplace. Worse, even existing proposals lack any policy for *runtime adaptation* of EE configurations, i.e., the set of active ramps and their thresholds. Such adaptation is crucial since dynamic workload characteristics govern the efficacy of each ramp in terms of exiting capabilities and added overheads (to non-exiting inputs); failure to continually adapt configurations can result in unacceptable accuracy drops up to 23.9% for our workloads. However, devising adaptation policies is difficult: the space of configurations is massive, and it is unclear how to obtain a signal for accuracy monitoring once an input exits.

We present **Apparate**, the first system that automatically (i.e., without developer effort or expertise) injects and manages EEs for serving with a wide range of models. Our main insight is that the above challenges are not fundamental to

[*] Equal contributions.

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/10.1145/3694715.3695963

EEs, and instead are a byproduct of what we are trying to get out of them. Specifically, adaptation challenges largely stem from halting execution for an input upon an exit, which leaves uncertainty in the 'correct' response (as per the non-EE model). Instead, Apparate uses EEs *only to deliver latency reductions*; results for successful exits are immediately released, but *all* inputs continue to the end of the model. The key is in leveraging the (now) redundant computations to enable continual and efficient adaptation, while also remaining compatible with proven compute efficiency optimizations such as batching and model compression.

Guided by this philosophy, Apparate runs directly atop existing serving platforms and begins by automatically converting registered models into EE variants. Apparate's EE preparation strategy must strike a balance between supporting fine-grained runtime adaptation without burdening those time-sensitive algorithms with (likely) unfruitful options. To do so without developer effort, Apparate leans on guidance from the original model design, crafting ramp locations and architectures based on downstream model computations and data flow for intermediates around the model. Original model layers (and weights) are unchanged, and added ramps are rapidly trained in parallel (for efficiency), but in a manner that preserves their independence from other ramps.

Once deployed, Apparate continually monitors EE operation in GPUs, tracking computations and latency effects of each ramp, as well as outputs of the original model (for accuracy ground truth). To tackle the massive space of configuration options, Apparate judiciously decouples tunable EE knobs: thresholds for existing ramps are frequently and quickly tuned to ensure consistently high accuracy, while costlier changes to the set of active ramps occur only periodically as a means for latency optimization. For both control loops, Apparate leverages several fundamental properties of EEs to accelerate the tuning process. For instance, the monotonic nature of accuracy drops (and increases in latency savings) for higher thresholds motivates Apparate's greedy algorithm for threshold tuning which runs 3 orders of magnitude faster than grid search while sacrificing only 0–3.8% of the potential latency wins.

We evaluated Apparate across a variety of recent CV and NLP models (ranging from compressed to large language models), diverse workloads (classification and generative), and several serving platforms (TensorFlow-Serving [51], Clockwork [30], and HuggingFace Pipelines [6]). Compared to serving without EEs, Apparate improves 25th percentile and median classification latencies by 70.2–94.2% and 40.5–91.5% for CV, and 16.0–37.3% and 10.0–24.2% for NLP, while imposing negligible impact on platform throughput. Latency wins are similar for generative scenarios: 22.6–77.9% median time-per-token improvements. Importantly, unlike existing EE proposals that yield accuracy dips up to 23.9%, we find that Apparate's adaptation strategies *always* met set accuracy constraints. We open source Apparate at https://github.com/dywsjtu/apparate.

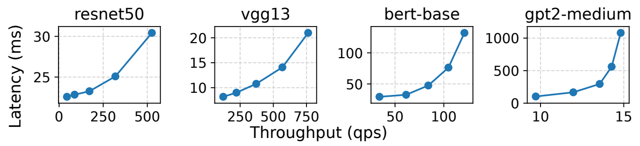


Figure 1. Throughput-latency tradeoff in model serving. Results show serving times with batch sizes of 1–16.

2 Background and Motivation

We start by overviewing existing ML serving platforms (§2.1), highlighting the challenges they face in balancing metrics that are important for system performance (i.e., throughput, resource utilization) and application interactivity, i.e., per-request latencies. We then describe the promising role that early exits can play in alleviating those tensions (§2.2), and the challenges in realizing those benefits in practice (§2.3). Results follow the methodology from §4.1.

2.1 Model Serving Platforms

ML models are routinely used to service requests in interactive applications such as real-time video analytics [15, 65], chatbots [71], recommendation engines [66], or speech assistants [13]. To manage such workloads, especially at large scale, applications employ serving platforms such as ONNX runtime [8], TensorFlow-Serving [51], PyTorch Serve [11], Triton Inference Server [7], among others [20, 30, 44, 58, 65, 80]. These platforms ingest pre-trained model(s), often in graph exchange formats like ONNX [9] and NNEF [2], and are granted access to a pool of compute resources (usually with ML accelerators such as GPUs) for inference.

Given the latency-sensitive nature of interactive applications, requests are often accompanied with *service level objectives* (SLOs) that indicate (un)acceptable response times for the service at hand. In particular, responses delivered after an SLO expires are typically discarded or yield severely degraded utility. Common SLOs are in the 10–100s of milliseconds, e.g., for live video analytics [52, 65].

During operation, serving platforms queue up incoming requests that can arrive at fixed or variable rates, and continually schedule jobs across the available compute resources. An inference task may be scheduled to run on a single node in a cluster, or may be distributed across multiple nodes [30, 80].

Latency-Throughput tension. To support the need for high *throughput*, serving platforms resort to *batching*, whereby inputs are grouped into a single high-dimensional tensor that moves through the model in lockstep, kernel by kernel, with final per-request responses being delivered at the same time. Larger batch sizes amortize the cost of loading a kernel into GPU memory across more inputs, and enable more effective use of accelerator parallelism [20, 82].

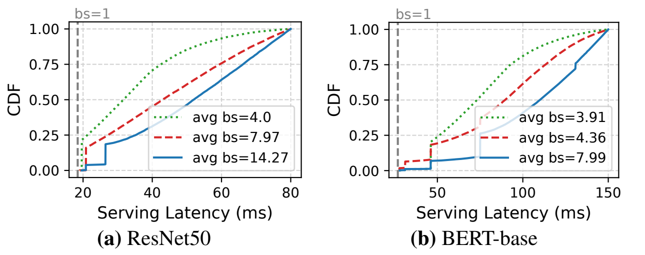


Figure 2. Tuning platform knobs lowers latencies but harms throughput. Results vary TF-Serve's *max_batch_size* from 4–16. Gray lines show min serving time per model (batch=1). CV uses a random corpus video; NLP uses Amazon reviews [1].

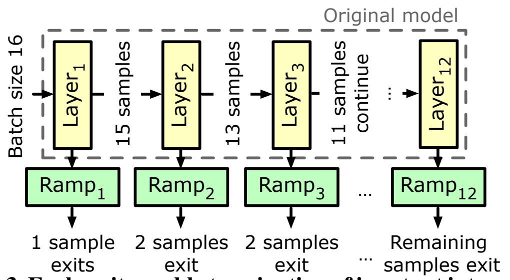


Figure 3. Early exits enable termination of inputs at intermediate layers, lowering both compute and latency.

Unfortunately, delivering the throughput necessary to support high request rates [32, 50] is directly at odds with *per-request* latencies (Figure 1). On one hand, latency for an input is minimized by scheduling inference as soon as the request arrives with batch size of 1. On the other hand, throughput is maximized by creating large batches using a queuing system which directly inflates request latencies.

The problem. In navigating this tension, the key decision that serving platforms face is when to drain queued requests for inference. Certain platforms [20, 30, 65] take an all-or-nothing stance on latency, with adherence to SLOs considered complete success, and violations viewed as failure. Accordingly, these platforms schedule inference jobs in a work-conserving manner and select the *max* batch size that limits SLO violations for queued requests. However, many interactive applications present a more nuanced latency story where sub-SLO responses are not equally useful, e.g., faster responses boost conversational interactivity for chatbots [33, 79] and confidence in scene perception for video analytics [15, 67].

Other platforms [7, 11, 51] provide more flexibility by exposing tunable knobs to guide queue management, e.g., *max_batch_size* and *batch_timeout_micros* parameters cap batch sizes or inter-job scheduling durations. However, such knobs do little to ease the throughput-latency tension, presenting harsh tradeoffs (Figure 2): tuning for median latency improvements of 17.3–39.1% brings 1.1–3.6× reductions in average batch sizes (and proportional hits on throughput).

Platforms for serving generative models [44, 80] face similar tensions despite the less explicit focus on SLOs (since sequence lengths are hard to predict). Indeed, although such platforms use continuous batching to ensure that new requests

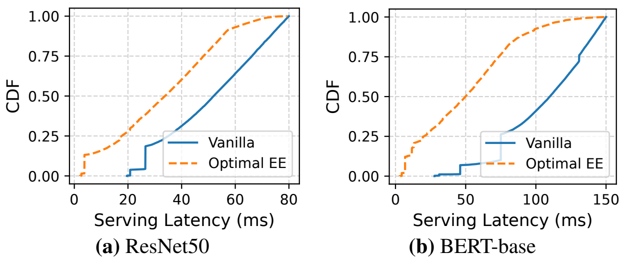


Figure 4. EEs can lower latencies without harming throughput. Results modulate latencies from TF-Serve with original/vanilla models (Figure 2) based on optimal exiting.

immediately leverage idle resources as any input's generation finishes, they prioritize throughput by running at the highest possible batch size (capped by a preset max).

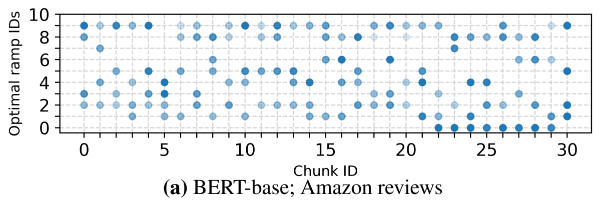
Takeaway. Existing platform configurations and knobs fail to practically remediate the throughput-latency tension, and instead simply navigate (often) unacceptable tradeoff points between the two goals. Given ever-growing request rates and the need for high throughput, we ask if there is a middleground: whereby new serving adaptations enable lower per-request latencies (moving closer to the lower-bound serving times in Figure 2) without harming platform throughput.

2.2 Early-Exit Models

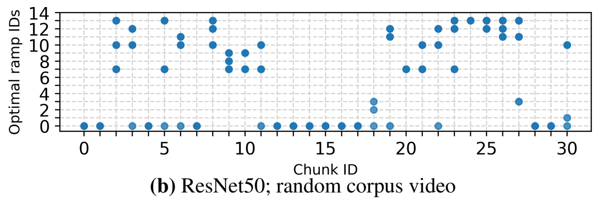
Early (or multi) exit models [69, 76] present an alternate way to address this tension by rethinking the granularity of inference. As shown in Figure 3, the key premise is that certain 'easy' inputs may not require the full predictive power of a model to generate an accurate result. Instead, results for such inputs may be predictable from the values at intermediate layers. In such cases, the foregone model execution can yield proportional reductions in both per-request latencies and compute footprints. Thus, the goal with early exits (EEs) is to determine, on a per-input basis, the earliest model layer at which an accurate response can be generated.

To use EEs, intermediate layers in a model are augmented with *ramps* of computation. These ramps ingest the values output by the layers they are attached to and parse them to predict the final model's result, e.g., a classification label. Ramps can perform arbitrary degrees of computation to arrive at a potential result. Exiting decisions at each ramp are made by comparing the entropy in the predicted result (or averaged over the past *k* ramps) to a preset *threshold*. Thresholds are set to balance latency and compute wins with potential dips in accuracy; a higher threshold implies lower required confidence for exiting, and thus more exiting.

Potential benefits. To understand the effect that EEs can have on the latency-throughput tension, we used off-the-shelf EE variants for the models in Figure 2: BranchyNet [69] (CV) and DeeBERT [76] (NLP). For each model-input pair, we identified the *optimal* exit point defined as the earliest exit ramp that predicted the correct response for the input. We then modified the highest-throughput results in Figure 2 to account for exiting by subtracting the time saved for each



(a) BERT-base; Amazon reviews



(b) ResNet50; random corpus video

Figure 5. Optimal EE configurations change frequently. Workloads use 64-request chunks. Dot presence shows a ramp that was part of the optimal config for a chunk, while transparencies indicate threshold values (opaque is higher).

Strategy\Workload	CV	NLP
Initial Only	84.5% (74.3%)	86.8% (73.6%)
Uniformly Sampled	90.3% (64.2%)	87.7% (69.4%)
Continual Tuning	98.6% (43.5%)	98.3% (26.6%)

Table 1. Thresholds need frequent tuning to avoid accuracy loss. Continual tuning kicks in when chunk accuracy < 99%. Results list avg accuracy (median latency wins).

exiting input, i.e., the difference in time for passing an input to the end of its optimal ramp versus passing it to the end of the model (without any ramps). These results are conservative upper bounds in that they do not reduce queuing delays or alter job scheduling. As shown in Figure 4, without changing queueing decisions, EEs can bring 35–54.7% and 17.9–26% improvements in median and 95th percentile latencies.

2.3 Challenges

Despite many EE proposals from the ML community [14, 35, 48, 60, 61, 69, 76, 77, 84], and their potential benefits, multiple issues complicate EE use in practice, limiting adoption.

C1: Latency and resource overheads. Although exiting can enable certain inputs to eschew downstream model computations, exit ramps impose two new overheads on serving. First, to be used, ramps must also be loaded into GPU memory which is an increasingly precious resource as models grow in size [40, 64, 80] and inference spreads to resource-constrained settings [31, 52]. For instance, DeeBERT inflates memory requirements by 6.6% compared to BERT-base. Second, certain inputs may be too "hard" to accurately exit at an intermediate ramp. In these cases, serving latency and throughput mildly worsen as unsuccessful exiting decisions are made, e.g., inputs that cannot exit at any ramp slow by 22.0% and 19.5% with BranchyNet and DeeBERT.

C2: Frequent and costly adaptation. As shown in Figure 5, the evolving nature of workloads for interactive applications brings frequent changes in the best EE configuration at any time, i.e., the set of active ramps (and their thresholds) that maximize latency savings without sacrificing response accuracy. Unfortunately, the large body of EE literature is unaccompanied by any policy for tuning ramps and thresholds during serving. Instead, proposed EE models are equipped with the max number of ramps, and mandate users to perform one-time tuning of thresholds. Such tuning is non-trivial and fails to cope with workload dynamism. For example, Table 1 shows how one-time tuning on sampled data brings 8.3–14.5% drops in accuracy relative to continual tuning. Worse, the space of configurations is untenably large, with many ramp options (i.e., at any layer, with any computation) and a continuous space of possible threshold values for each.

C3: Lack of accuracy feedback. EE ramp decisions are ultimately confidence-driven and may result in accuracy degradations (as shown above). In production scenarios, serving optimizations that deliver accuracy reductions >1–2% are generally considered unacceptable [16]. Yet, once deployed, EE models do not provide any indication of accuracy drops; indeed, when an exit is taken, the corresponding input does not pass through the remaining model layers, and the original (non-EE) model's prediction is never revealed. Thus, with early exiting, we lack mechanisms to determine when accuracy degradations are arising and EE tuning is required.

3 Design

Apparate is an end-to-end system that automatically integrates early exits into models and manages their operation throughout the inference process. Its overarching goal is to optimize per-request latencies while adhering to tight accuracy constraints and throughput goals. Our key insight is in rethinking the way that EEs are configured and the benefits they are expected to deliver. In particular, rather than using EEs in the traditional way – where inputs exit model inference to provide *both* latency and computational benefits – Apparate focuses solely on latency savings by allowing *results* to exit, with inputs still running to completion. Foregoing true exiting (and thus, compute savings) grants Apparate with direct and continual feedback on EE accuracy (C3). This feedback provides the requisite signals for Apparate to continually adapt EE configurations to maximize latency savings while catering to resource constraints and workload dynamics (C1, C2).

Apparate's design represents a departure from the typical expected utility of EEs (i.e., compute savings) that has been fraught with practical challenges. Instead, Apparate demonstrates an alternate avenue for benefits that EEs can bring (i.e., latency reductions), while remaining compatible with other compute efficiency optimizations that have had substantial practical traction. For instance, by foregoing true exiting, Apparate can run alongside request batching [39]. Further, Apparate supports diverse model architectures, including those that have been compressed for efficiency (§4.2). We note that the redundant computations in Apparate match the work that vanilla serving perform by executing all model layers.

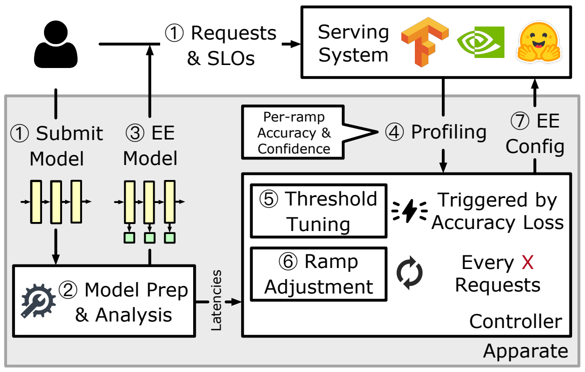


Figure 6. System architecture.

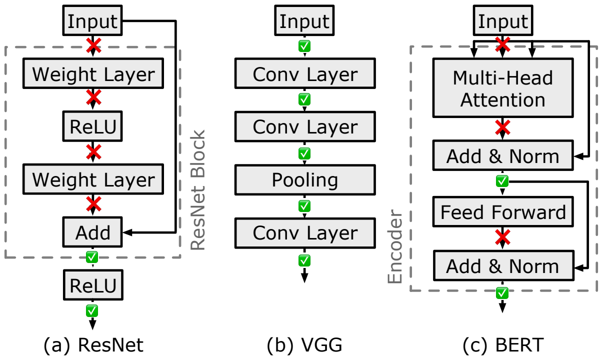


Figure 7. Apparate only injects ramps that make full use of available data flows at that part of the model.

Figure 6 overviews Apparate's workflow, which runs atop existing serving platforms. Users register inference jobs as normal ❶, providing models and SLOs without needing any awareness of or expertise about EEs. In addition, Apparate introduces two parameters: (1) *ramp aggression*, which bounds the number of active ramps in terms of % impact on worst-case latency (and throughput), and (2) *accuracy constraint* which indicates how much (if any) accuracy loss is acceptable relative to running the submitted model on all inputs without exiting. Apparate's controller begins by configuring the provided model with EEs ❷, performing a graph assessment to determine suitable positions for ramps, and training those ramps on bootstrap data (§3.1). The resulting model is passed to the serving platform for deployment ❸, after which Apparate shifts to management mode. In this phase, as requests arrive and inference is scheduled, Apparate's controller gathers real-time feedback on the utility of each ramp (overheads vs. latency savings) and achieved accuracies (relative to the original model) ❹. This data is used to continually adapt the EE configuration ❼ at different time scales: rapid threshold tuning for accuracy preservation (§3.2) ❺, and less frequent ramp adjustments for latency optimization (§3.3) ❻.

This decoupling of EE configuration adaptation into two tuning control loops is a key design decision that Apparate uses to manage the untenably large search space of ramp-threshold combinations (C1) without substantial loss in EE efficacy. Specifically, Apparate chooses to use frequent threshold tuning to preserve accuracy because it provides a finer-grained knob for walking the accuracy-latency tradeoff, i.e., thresholds are continuous, whereas ramp locations are inherently discretized. Thresholds also provide a mechanism to control ramp location; at the extreme, thresholds for any active ramps can be tuned to preclude exiting. Regardless, to limit foregone wins from infrequent ramp tuning, Apparate opts to employ many lightweight ramps (§3.1): even if an optimal ramp is not present yet, a nearby ramp is likely active and can provide much of the same wins.

Implementation details. Apparate is implemented as a layer atop existing serving platforms (currently three [6, 30, 51], though its techniques generalize to others), and comprises ~7500 lines of Python code for EE preparation (§3.1) and management (§3.2-3.3). Apparate runs a separate controller per model replica (as decided by the serving platform) on a CPU, with GPUs streaming per-ramp/batch profiling information in a *non-blocking* fashion. This is possible since inputs pass to the end of models with Apparate, irrespective of exiting decisions. Tasks associated with model handling and serving are handled by the underlying serving platform, e.g., loading from disk, queuing, and batching.

3.1 Preparing Models with Early Exits

Upon job registration with any DNN, Apparate's initial task is to automatically prepare the model to leverage EEs without developer effort. This phase repeats any time the submitted model changes, e.g., continual retraining [15, 42, 67]. Note that, in the event that a developer provides an EE model, Apparate can forego any training and instead immediately begin managing its exit configurations (§3.2–3.3).

Ramp locations. Apparate accepts a model in the ONNX format, a widely used IR that represents the computation as a directed acyclic graph [9]. Once ingested, Apparate must first identify candidate layers for ramp addition. The goal is to maximize ramp coverage across the model (to provide more configuration options for Apparate's runtime management), while avoiding ramps that are unlikely to be fruitful (but add complexity to adaptation decisions). To balance these aspects for diverse models, Apparate marks feasible ramp locations as those where operators are *cut vertices*, i.e., a vertex whose removal would disconnect a graph into two or more disjoint sub-graphs. In other words, no edge can start before a ramp and re-enter the model's computation after the ramp.

The idea is that such ramps take advantage of all available data outputs from the original model's processing to that point, boosting their chance at accurate predictions. As an example, consider families like ResNet or BERT which enable deep models by stitching together series of residual blocks, i.e., ResNet blocks for convolutions, or BERT encoders that each embed multi-head attention and feed-forward network residual blocks. To avoid performance degradations late in the model, the output of each block is ultimately a combination of its processing results and its input. In such scenarios, Apparate injects ramps between blocks, but not within each

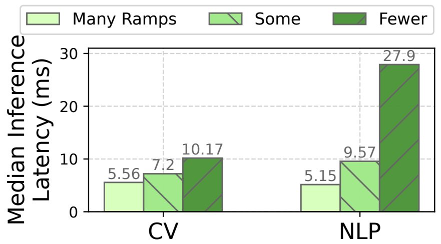


Figure 8. More lightweight ramps boost EE savings. Results compare Apparate's default 'many ramps' with versions that use fewer, more expensive ramps.

block to avoid ramps making decisions on partial data, i.e., ignoring block inputs. Ramps are similarly injected between trasnformer blocks of generative models, but only for decoding phases (as input tokens must be fully processed for generation). In contrast, for VGG models, ramps are feasible at all layers since their intermediates represent the full extent of data flow throughout the model. Figure 7 depicts examples.

Overall, this strategy results in 9.2–68.4% of layers having ramps for the models in our corpus, which we empirically observe is sufficient to adapt to dynamic workloads (§4.2). However, we note that Apparate can directly support any other ramp configuration strategy, and offers a simple API for developers to express ramp policies or restrictions.

Ramp architectures. For each feasible ramp location, Apparate must determine the style of ramp computations to use. Recall from §2 that ramps can ultimately be composed of arbitrary layers and computations, with the only prerequisite being that the final layer sufficiently mimics that of the original model to ensure that response formats match. Determining the appropriate ramp complexity in this large space presents a tradeoff: additional computation can improve the exit capabilities of a ramp, but comes at the expense of (1) increased ramp latency, and (2) coarser flexibility and coverage at runtime since ramps become illogical if their computation exceeds that in the original model up until the next ramp.

Apparate opts for the shallowest ramps that can transform the intermediates at any layer into a final model prediction. Specifically, ramps comprise the model's final fully-connected (fc) layer, prepended with a lightweight pooling operation that reduces the dimensionality of intermediates to ensure compatability with the fc layer. This manifests differently for various model types. For instance, for vision models like ResNet, pooling is simply the model's penultimate layer. Similarly, for generative LLMs, ramps can simply use the final decoder head to transform intermediate hidden states. In contrast, for BERT, only the basic operator is drawn from the BERT pooler module, i.e., extracting the hidden state corresponding to the first token [23]. For all models, the input width of the fc layer is modified to match the intermediates at each ramp location; the output remains unchanged to preserve result formats.

Figure 8 evaluates this methodology by comparing with two, more expensive alternatives. With ResNet, to mimic model operations following each ramp, we add 1–2 convolution layers prior to pooling. For BERT, we consider two approaches: (1) add two fc layers after pooling, each with reduced width to shrink inputs to the final fc, and (2) following DeeBERT [76], replace the simple pooling operator with the entire BERT pooler block and add a dropout as in the original model. In all cases, the number of ramps is subject to the same budget (i.e., Apparate's default uses the most ramps), ramps are uniformly spaced across feasible positions in each model, and thresholds are optimally selected as in §2.2.

We observe that the added compute has minimal effect on ramp efficacy. For example, median latencies are 1.3–1.8× and 1.9–5.4× smaller with Apparate's default ramps than the complex alternatives for CV and NLP. Nonetheless, to show Apparate's generality, we consider other ramp styles in §4.5.

Training ramps and deploying models. To determine the appropriate weights for each ramp, Apparate can automatically label a dataset that is either developer-provided or collected online (running the vanilla model on live inputs). Automatic labeling is feasible since ramps aim to mimic the submitted model's behavior (not ground truth), so the submitted model's outputs can directly serve as labels. Regardless, during training, Apparate freezes the original model weights to ensure that non-EE behavior and feedback for tuning EEs is unchanged from the user's original intentions. Since its ramps are lightweight (single fully-connected layers) and comprise only 0.01–3.50% of our models' parameters, the FLOPs required for Apparate's ramp training is significantly lower than whole-model pre-training or even fine-tuning. In cases where existing (final) layers can be used as ramps, e.g. in generative scenarios, Apparate eschews training and directly reuse the final layer for each ramp. In addition, Apparate enforces that *all* inputs are used to train all ramps, i.e., exiting is prohibited during training. This ensures that ramps are trained *independently* of the presence (or behavior) of any upstream ramps, which is crucial since the set of active ramps can vary at runtime. Further, such independence and model freezing enable loss calculations to be backwards propagated *in parallel* across ramps, rapidly speeding up training despite the many lightweight ramps. As a result, ramp training only takes on the order of a few minutes for our models using a single A6000 GPU. Apparate uses the first 10% of each dataset for training and validation (following a 1:9 split).

For initial deployment, Apparate evenly spaces the max number of allowable ramps across the model. To avoid accuracy dips due to discrepancies between training data and the current workload, each ramp begins with a threshold of 0, i.e., no exiting. The updated model definition (with enabled ramps) is passed to the serving platform which runs normally.

3.2 Accuracy-Aware Threshold Tuning

To avoid accuracy drops as workload characteristics change over time, Apparate's controller employs frequent and fast tuning of thresholds for already-enabled ramps. The reason is that threshold tuning for any set of ramps is sufficient to ensure that user-specified accuracy constraints are not violated – at the extreme, all thresholds could be set to zero, which precludes any early exiting. Altering only the set of active ramps fails to provide this property.

To enable threshold tuning, as requests pass through a model, Apparate continually records exiting information at each active ramp, as well as the final result that the original model predicts. More precisely, Apparate records the highest-confidence result for each ramp, even if the error exceeds the ramp's threshold (precluding exiting). Importantly, since inputs always pass fully through models with Apparate, this information is recorded for all inputs at each active ramp, irrespective of upstream exiting decisions. This is paramount since the information serves not only signals when to tune thresholds, but also provides guidance for how to do so.

Triggering tuning. Apparate maintains an average achieved accuracy over the past 16 samples by comparing exiting results with the deployed configuration to results of the original model. Threshold tuning is triggered any time a window's accuracy falls below the user-specified accuracy constraint. The threshold tuning process (described below) runs asynchronously on a CPU, without any disruptions to ongoing jobs. This is possible since thresholds are anyway enforced only by Apparate's controller; GPUs are agnostic to threshold values, and instead simply stream ramp results to the Apparate controller which determines exiting decisions.

Evaluating threshold configurations. Threshold tuning needs insight into how any alterations to active ramp thresholds would affect model exiting behavior (and accuracies). By observing per-request behavior *only at active ramps*, Apparate can rapidly evaluate any threshold configuration without additional inference, and while accounting for inter-ramp dependencies. In particular, to evaluate new threshold values, Apparate simply identifies the earliest ramp whose top prediction now has an error rate below its threshold. Comparing these results with those of the original model indicates the achieved accuracy for the new configuration; latency wins are computed using the one-time profiling data described in §3.3.

Greedy search. The goal of tuning is to identify a new set of thresholds that maximizes latency savings while adhering to accuracy constraints for the last window of data. The challenge is that the space of thresholds to consider is massive, precluding a grid search (especially given how frequently adaptation is needed - §2.3). Indeed, even with discretized threshold values in [0, 1] with a step size of S, computation costs are $O(C \times (\frac{1}{S})^R)$, where R is the number of active ramps, and C is the cost to evaluate a given configuration.

Instead, Apparate employs a greedy heuristic that leverages a fundamental property of EEs when evaluated against an original model: *higher thresholds result in monotonic decreases in accuracy and monotonic increases in latency savings*. This prunes the space of threshold values to consider by providing a clear boundary in the R-dimensional space that separates configurations that are sufficiently accurate from those that are not. Additionally, for accurate configurations, maximum latency savings must fall on that boundary. Figure 9 illustrates this.

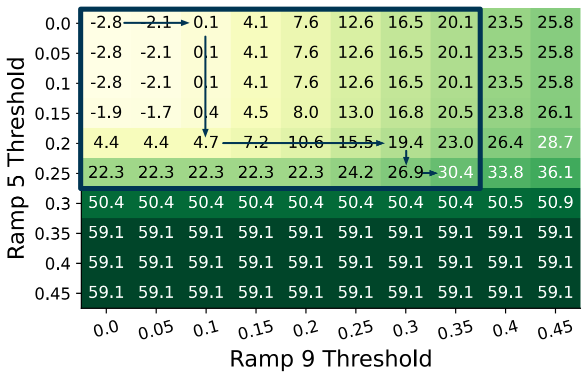


Figure 9. Threshold tuning example with two active ramps for ResNet50 and a random video. Configurations within the boundary have <1% accuracy loss; cell values list latency wins. Arrows show the path taken by Apparate's hill climbing algorithm.

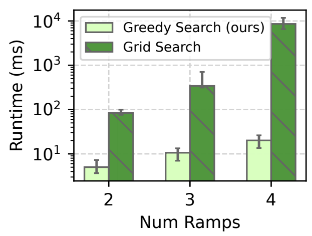
These properties inform Apparate's hill climbing strategy [63] for threshold tuning. Starting with threshold values of 0 for each active ramp, and a step size of 0.1, threshold tuning runs in a series of (incremental) exploration rounds. In each round, we increase the threshold of each ramp in isolation (leaving the others unchanged), and evaluate the achieved accuracy and latency savings as described above. Apparate then chooses the single ramp threshold change that delivered the largest additional latency savings per unit of additional accuracy loss. This process repeats until no ramp's threshold can be increased without an accuracy violation.

To enhance this process, Apparate follows a multiplicative increase, multiplicative decrease policy on step sizes to balance search speed and granularity. Each time a step increase results in an accuracy violation, Apparate halves that ramp's step size for subsequent rounds to hone in on the boundary at fine granularity; step sizes are lower-bounded at 0.01. Conversely, selection of a ramp for threshold alteration suggests a promising path of exploration; in this case, for a speedup, Apparate doubles that ramp's step size for the following round.

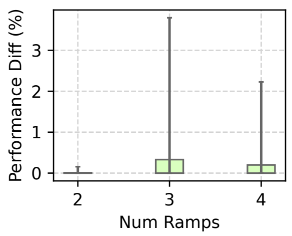
Overall, as shown in Figure 10, Apparate's threshold tuning algorithm runs up to 3 orders of magnitude faster than a pure grid search (11.9ms vs. 3.0s on average). Note that these results maximally parallelize grid search across a 32-core machine. Further, selected threshold values achieve within 0–3.8% of the latency savings of the optimal configurations.

3.3 Latency-Focused Ramp Adjustments

The set of active ramps ultimately dictates where inputs can exit, and thus provides bounds on potential latency savings. Unlike threshold tuning which runs reactively (since accuracy is a constraint) and uses only recent profiling data to evaluate new configurations, ramp adjustment is used strictly as an



(a) Threshold tuning speed.



(b) Optimality of tuning results.

Figure 10. Apparate's tuning vs. optimal tuning on runtime and latency of selected configurations. Bars list medians across all model-workload pairs, with error bars for min-max.

optimization (for latency savings) in Apparate, and requires deployment to evaluate the impact of any new ramp. Thus, Apparate's ramp tuning runs periodically (every 128 samples by default) and conservatively alters the set of active ramps to incrementally converge on high-performing configurations.

Evaluating active ramps. In each round, Apparate's controller starts by computing a utility score for each active ramp that evaluates its overall impact on workload latency. To do so, Apparate couples per-ramp exit rates (§3.2) with two additional inputs that are collected once per model during bootstrapping: (1) the latency overhead per ramp, and (2) a layer-wise breakdown of time spent during model inference (for different batch sizes [30] and including network delays for distributed serving). The latter is necessary since latency characteristics vary across models but govern the impact of exits, e.g., latency arises early in CV models [52], but more evenly across coding blocks in transformers. Using these inputs, Apparate defines the utility of ramp R as `savings - overheads`, where `savings` is the sum of raw latency that exiting inputs avoided by using ramp R, and `overheads` is the sum of latency that R added to inputs that it could not exit.

Adding new ramps. If any negative utility values exist, Apparate applies a fast threshold tuning round to see if ramp utilities become entirely positive without harming overall latency savings. If not, Apparate immediately deactivates all negative-utility ramps. From there, the key question to address is what ramps (if any) should be added to make use of the freed ramp budget. The main difficulty is in predicting the utility of each potential addition. Indeed, while per-exit latency savings for each potential ramp are known (using the latency breakdown from above), exit rates are not.

To cope with this uncertainty, our guiding intuition is that, subject to the same accuracy constraint, *later ramps almost always exhibit higher exit rates than earlier ones*. The reason is that late ramps have the luxury of leveraging more of an original model's computations when making a prediction. Importantly, this implies that a candidate ramp's exit rate is bound by the exit rate of the closest downstream ramp; note that this is not a formal guarantee [41], and Apparate uses this for search efficiency (not correctness).

Building on this, Apparate's controller computes an upper bound on the utility of candidate ramps as follows. To

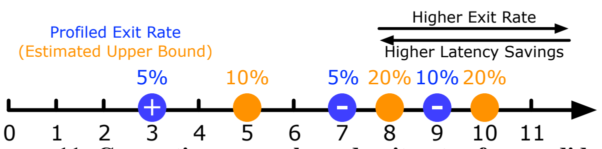


Figure 11. Computing upper-bound exit rates for candidate ramps. Blue dots show previously active ramps (+/− indicates positive/negative utility), while orange dots show candidates.

avoid inter-ramp dependencies harming ramps that are already performing well, we only consider additions after the latest positive ramp P in the model. In particular, Apparate divides the range following P into intervals separated by any negative ramps deactivated in this round. The first round of candidate ramps are those in the middle of each interval.

For each candidate ramp, we compute its upper-bound exit rate as the sum of profiled exit rates for the following deactivated ramp and any earlier deactivations (Figure 11); the idea is that inputs from earlier deactivations would have reached the following deactivated ramp and *might* have exited there. Utility scores are then computed as above, and the ramp with the highest positive utility score is selected for trial. If all ramps have negative projected utilities, Apparate repeats this process for later candidate ramps in each interval. Once a ramp is selected for trial, Apparate adds it to the deployed model definition, while removing deactivated ramps. Trialed ramps start with threshold=0 to prevent inaccurate exiting, but are soon updated in the next round of threshold tuning.

Until now, we have only discussed how Apparate handles scenarios with at least one negative ramp utility. In the event that all ramps exhibit positive utilities, Apparate enters a low-risk probing phase to determine if latency savings can grow by using earlier ramps. If ramp budget remains, we add a ramp immediately before the existing ramp with *highest* utility (while keeping that ramp to preserve its exiting wins). If not, we shift the ramp with the *lowest* utility score one position earlier, leaving the most positive ramp untouched.

3.4 Supporting Generative Large Language Models

Recent efforts have incorporated EEs into generative language models (e.g., CALM [60] and FREE [14] for T5 [56], LayerSkip [24] for Llama [71]) to enhance their response times in interactive applications. While the general challenges with EEs persist – lack of runtime adaptation for time-varying accuracy and latency savings – the auto-regressive nature of generative models poses a unique challenge for Apparate. Unlike with classification where inputs can be processed to completion *independently*, each token generated by an LLM depends on the preceding ones in the sequence, requiring their full key-value (KV) states. Thus, when a token $T1$ exits using a ramp, the generation delay begins to accumulate for the next token, $T2$; yet, $T2$'s forward pass through the layers after the ramp cannot begin until $T1$ passes through the same layers to generate KV states. The effect is potential harm on time-per-token (TPT) distributions.

To regain latency savings from EEs, Apparate draws inspiration from recent parallel decoding techniques [14, 24, 45]. As a token exits from a ramp, Apparate *does not immediately compute the remaining model layers*, and instead only accumulates its hidden states at that ramp. The remaining computations are executed in parallel alongside the first non-exiting token that is encountered thereafter at that ramp. For concreteness, consider a scenario with 2 tokens, $T1$ and $T2$. Assume $T1$ can exit at ramp $R1$, while $T2$ must proceed later in the model. $T1$ will exit at $R1$, and $T2$ will immediately begin processing. However, once $T2$ fails to exit at $R1$, the remaining layers for $T1$ are run alongside (i.e., batched with) $T2$'s remaining layers. Taken together, $T1$ achieves per-token latency savings from exiting, while $T2$ incurs a very mild (§4.3) penalty from the higher batch size.

In addition to improved TPT latencies and compute efficiency (due to batching effects on GPUs), parallel decoding grants Apparate token-level feedback for exiting decisions relative to the original (non-EE) model. Specifically, for each parallel decoding instance, Apparate collects per-token feedback up until the first token whose exit result deviates from the original model; feedback for subsequent tokens is discarded as it may reflect cascading errors from inter-token dependencies. This feedback guides Apparate's threshold and ramp tuning strategies from §3.2–3.3.

4 Evaluation

We evaluated Apparate across a wide range of NLP and CV workloads and serving platforms. Our key findings are:

- Apparate lowers 25th percentile and median classification latencies by 40.5–91.5% and 70.2–94.2% for CV, and 16.0–37.3% and 10.0–24.2% for NLP workloads, compared to original (non-EE) models. These wins are 5.7–66.6% larger than two-layer inference systems using compressed models for 'easy' inputs [17, 73]. Median time-per-token wins are 22.6–77.9% for generative scenarios.
- Unlike existing EE models that unacceptably worsen accuracies and tail latencies by up to 23.9% and 11.0% for classification, Apparate consistently meets accuracy and tail latency constraints. This carries to generative scenarios: Apparate's tail latency is 1.8–2.4% lower than existing EE models which lower accuracies up to 5.5%.
- Apparate automatically generalizes to different model architectures (e.g., compressed) and EE configurations (e.g., ramp style), and its wins gracefully shrink as accuracy or tail-latency constraints grow.

4.1 Methodology

Models. For *classification*, we consider 10 models (across 4 families) that cover popular architectures and diverse sizes. For CV, we use the ResNet{18, 50, 101} residual models, as well VGG{11, 13, 16} models that follow a chained (linear) design. All of these models are pre-trained on ImageNet and from the PyTorch Model Zoo [55]. For NLP, we consider 3 encoder-only transformers from the BERT family – BERT-base, BERT-large, and Distilbert [59] (a variant of BERT-base that was compressed via distillation) – as well as a decoder-only LLM: GPT2-medium. These models span 66–345 million parameters, were collected from HuggingFace [38], and were pre-trained on Yelp reviews [10]. We also consider quantized versions of these BERT models in §4.2. For *generative scenarios*, we use the T5-large [56] encoder-decoder LLM (from prior EE work [14, 60]) with 770 million parameters and the Llama2 decoder-only LLMs with 7 and 13 billion parameters, both pretrained from HuggingFace.

Workloads. CV workloads comprise real-time object classification (people, cars) on 8 one-hour videos from recent video analytics literature [12, 34]. The videos were sampled at 30 frames per second, and span day/night from urban scenes.

NLP classification workloads focus on sentiment analysis using two datasets: Amazon product reviews [1] and IMDB movie reviews [53]. To the best of our knowledge, there do not exist public streaming workloads for NLP classification, so we convert these datasets into streaming workloads as follows. For Amazon, we follow the order of product categories in the original dataset, but within each category, we keep reviews only from frequent users (i.e., those with >1k reviews) and order streaming by user (250k requests in total). For IMDB, we follow the order of reviews in the original dataset, but stream each in sentence by sentence (180k requests in total). We then define arrival patterns using the Microsoft Azure Functions (MAF) as in prior work [30, 46]. To cope with the large variation in runtime across our models, we paired each model with a randomly selected trace snippet from the set that met the following criteria: (1) number of requests match that in our largest dataset, and (2) queries per second should not result in >20% dropped requests with vanilla serving for the given model and selected SLO (described below).

Our generative workloads use two datasets: CNN/DailyMail [62] for text summarization and SQuAD [57] for question answering. As in prior work [14, 60], we assign request arrival times that follow a Poisson distribution, configured to saturate our computing resources.

Parameter configurations. Given our focus on interactivity, we cope with heterogeneity in model runtimes by setting SLOs for classification to be 2× each model's inference time with batch size 1 in our main experiments. This results in SLOs between ~10–200 ms, which match the ranges used in prior work [30, 52, 65]; Table 5 in §7 lists the specific SLO values, and we study the effect of SLO on Apparate in §4.5. Unless otherwise noted, results use 1% for Apparate's accuracy constraint and a ramp budget of 2% impact on worst-case latency; we consider other parameter values in §4.5.

Setup. Experiments were conducted on dedicated servers with NVIDIA RTX A6000 GPUs housing 48GB of memory, AMD EPYC 7543P 32-Core CPUs, and 256GB DDR4

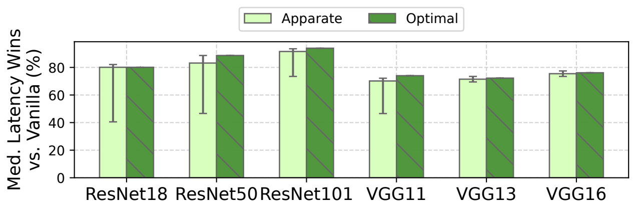


Figure 12. Median latency savings vs. vanilla models. Bars show median savings across all workloads; error bars are min-max.

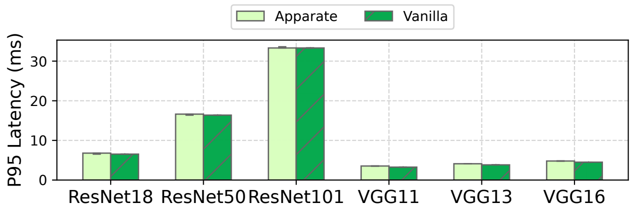


Figure 13. Evaluating Apparate's impact on tail latency (running with 2% budget) vs. vanilla serving. Bars show the median savings across all workloads with error bars spanning min-max.

RAM. We run experiments with two serving platforms for classification: TensorFlow-Serving [51] and Clockwork [30]. For space, we primarily present results with Clockwork, but note that reported trends hold for both platforms; we compare cross-platform results in §4.5. For generative workloads, we use the HuggingFace Pipelines inference engine [6].

Metrics and baselines. For classification, our main metrics are accuracy (% of inputs assigned correct label as per the non-EE model) and per-request response latency (including queuing). For generative models, sequence accuracy is measured using ROUGE-L (text summarization) and F1 (question answering) scores. Latency is measured using time-per-token (TPT) distributions. We compare Apparate with the following baselines: (1) original models without EEs (*vanilla*), (2) *optimal* EEs as defined in §2.2, i.e., assuming all inputs exit at their earliest possible ramps, with no ramp overheads, (3) two-layer inference systems [17, 73] that invoke full models only when compressed ones cannot generate high-confidence outputs, and (4) existing, non-adaptive EE strategies (§4.4).

4.2 Results for CV and NLP Classification

Figures 12–15 compare Apparate with vanilla model serving and optimal exiting across our workloads. Overall, Apparate significantly lowers latencies compared to serving vanilla models, while always adhering to the imposed 1% accuracy constraint. For instance, median speedups range from 40.5–91.5% (2.7–30.5 ms) for CV workloads, and jump to 70.2–94.2% (5.2–31.4 ms) at the 25th percentiles. NLP workloads follow a similar pattern, with median and 25th percentile savings ranging from 10.0–24.2% (3.9–25.3 ms) and 16.0–37.3% (4.8–53.2 ms). Importantly, across all workloads, Apparate's tail latency (and thus impact on throughput) always falls under its 2% budget, and is most often negligible (Figure 13).

Beyond this, there are several important trends to note. First, Apparate's raw latency savings grow with increased model sizes, e.g., 25th percentile wins of 53.2ms, 28.4ms, 14.3ms, 5.5ms for GPT-2, BERT-large, BERT-base, and Distilbert-base on the Amazon Reviews. This is because only results (not inputs) exit models with Apparate, so latency savings pertain entirely to serving times (not queuing delays) and grow as model size/runtime grows. Relative (%) latency savings follow the same pattern for CV workloads, e.g., Apparate's median wins grow by 13.8% and 5.3% moving from the smallest to the biggest models in the ResNet and VGG families. However, relative wins remain relatively stable in NLP models, e.g., 15.8% and 13.7% for GPT2 and BERT-large on Amazon Reviews. The difference is due to the effectiveness of the models in each domain. Results and task performance are largely similar across the CV models, enabling Apparate to inject ramps early in (even larger) models. In contrast, results are far better with the larger models in NLP; thus, Apparate's ramps fall in similar (relative) positions across the models.

Second, Apparate's wins are larger for CV workloads than NLP workloads for two reasons. As previously noted, CV workloads use lighter models and lower request rates (bound by video fps), and thus incur far lower queuing delays. More importantly, in contrast to CV where spatiotemporal similarities across frames (and thus, requests) are high due to physical constraints of object motion in a scene, NLP requests exhibit less continuity, e.g., back-to-back reviews are not constrained in semantic similarity. The effects on Apparate's adaptations are that (1) past data is less representative of future data, and (2) the duration until subsequent adaptation is shorter.

Other compute optimizations. To further illustrate Apparate's ability to run alongside existing compute efficiency optimizations – a key goal of its design (§3) – we ran experiments using post-training Int8 quantized Bert-base and Bert-large. Overall, we observe that Apparate's speedups largely persist, which median and 25th percentile wins of 7.3–19.4% and 6.9–31.1%. The mild dip in speedups relative to non-quantized Bert models (Figure 14) is a result of quantization's reduction in model overparameterization, which early exiting aims to capitalize on for select inputs.

Comparisons with optimal. As shown in Figure 12, latency savings with Apparate for CV workloads largely mirror those of the optimal that tunes exiting decisions based on perfect knowledge of the upcoming workload, e.g., median savings are within 20.5% of the optimal. In contrast, the limited continuity across inputs in NLP workloads leads to a wider gap of 65.4–78.5% at the median (Figure 15). To further characterize Apparate's performance on these workloads, we also consider a more realistic *online optimal* algorithm that relaxes the following elements. First, rather than per-sample adaptation of thresholds and ramps, ramp adjustments are set to operate only as fast as model definitions in the GPU can be updated. Second, rather than using perfect knowledge of upcoming inputs, decisions are made using only recent (historical) data; we tune based on the past {20, 40, 80} batches of inputs and select the one that performs best on the upcoming data. As

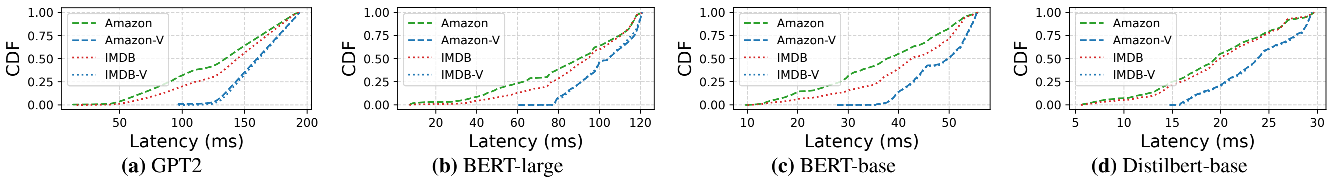


Figure 14. Apparate (with 2% budget) vs. vanilla models ("-V") on NLP classification workloads. "-V" curves per plot mostly overlap since they use the same timing trace and no exiting; minor discrepancies are due to the varying number of inputs across workloads.

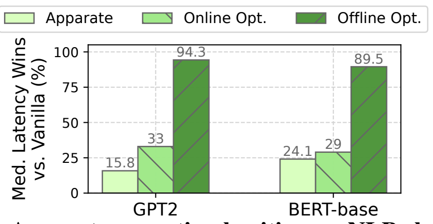


Figure 15. Apparate vs. optimal exiting on NLP classification workloads with the Amazon dataset.

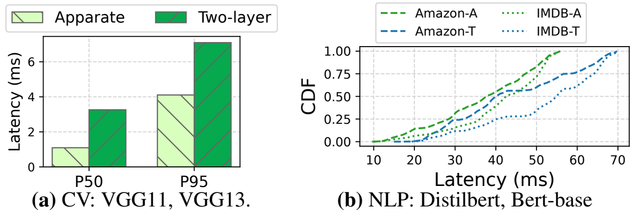


Figure 16. Apparate ("-A") vs. two-layer inference systems ("-T").

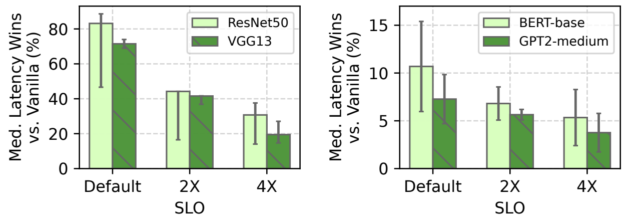


Figure 17. Impact of SLOs on Apparate's wins.

shown in Figure 15, Apparate's median latency savings are within 4.9–17.2% of this (more) realistic optimal strategy.

Comparisons with two-layer inference. We compare Apparate with in-house implementations of Tabi [73] (NLP) and FilterForward [17] (CV), which employ compressed models on all inputs, and pass only those inputs with low confidence results to the base model. Acceptable confidence scores from compressed models are configured to ensure both systems operate within the same accuracy loss budget as Apparate. Note that our evaluation is favorable to these systems: we ignore overheads from hosting the compressed models, compute overheads for data pruning (e.g., word pruning in Tabi), and queuing for batch formation between compressed and base models. As shown in Figure 16, across 3 representative workloads, Apparate delivers 5.7–66.6% and 20.9–42.0% lower median and P95 latencies compared to these baselines. For easy inputs, Apparate delivers latency wins by enabling ramps early enough (i.e., within the first third) in the base model to run faster than the baselines' compressed models. For hard inputs, tail latencies with Apparate are capped by its 2% budget; in contrast, the baselines add the entire compressed model runtime per input.

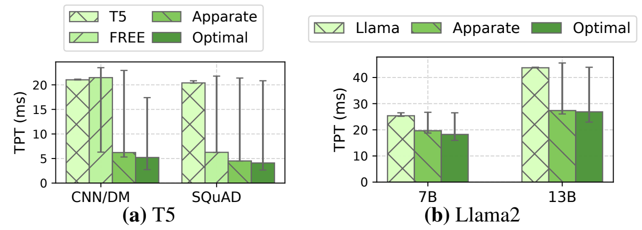


Figure 18. Comparison of Apparate against T5 generative LLM and FREE [14] (left), and against Llama generative LLM and its optimal variant (right). Bars represent medians with 25-95th percentiles.

Varying SLOs. We considered SLOs for each model that were 2× and 4× those in our default experiments (Table 5). Generally, higher SLOs induce larger serving batch sizes and higher per-request queuing delays; this dampens Apparate's relative latency savings (Figure 17) which target model runtimes, but not queuing delays. For instance, median latency savings for GPT-2 drop from 13.2% to 6.8% as SLO grows by 4×. Note that, to illustrate this trend for CV workloads, we upsampled each video to 120 fps. The reason is that our serving platforms are work-conserving and, at 30 fps, they are able to consistently schedule jobs with batch size 1 and low queuing delays given the low model runtimes.

4.3 Results for Generative Scenarios

Figure 18 compares Apparate with the vanilla T5-large LLM across our workloads. As shown, Apparate lowers median and 25th percentile TPT by 70.4–77.9% and 74.6–78.0%, respectively. However, at the 95th percentile, Apparate incurs 2.6–8.4% higher TPT due to ramp overheads and (more so) added delay from parallel decoding (non-exiting tokens run alongside the remaining computation for previously-exited tokens). Unlike with classification, formally bounding tail overheads online is hard because variable sequence lengths make TPT values unpredictable in vanilla generative serving. We also evaluated Apparate on the larger Llama models for question answering. Figure 18 shows that Apparate reduces the median and 25th percentile TPT by 22.6–37.4% and 25.4–40.3%. Notably, the latency wins increase as model size grows.

Comparison to optimal. Similar to classification, we define the optimal EE strategy as exiting each token at the earliest possible ramp that generates the correct value (ignoring delays

	Avg Acc	Median Wins	P95 Wins
Apparate (ResNet50)	99.0–99.2%	46.6–88.6%	-1.6–0.0%
BranchyNet	85.8–99.8%	-11.0–88.3%	-11.0%
BranchyNet+	76.1–99.9%	-11.0–88.3%	-11.0%
BranchyNet-opt	99.0–99.7%	-11.0–74.5%	-11.0%
Apparate (BERT-base)	99.1–99.3%	13.7–14.7%	2.1–3.0%
DeeBERT	91.7–97.1%	13.2–36.1%	-1.3–6.4%
DeeBERT+	82.2–90.3%	31.7–36.1%	5.9–6.4%
DeeBERT-opt	99.0%	9.8–36.1%	-1.4–6.4%

Table 2. Comparison with existing EE models. Results list accuracies and latency wins for all CV (top) or NLP (bottom) workloads. '+' and 'opt' use the optimized tuning from §4.4.

for generating the remaining KV states). Apparate's median TPT for T5 and Llama is within 9.0–16.2% and 2.1%–7.7% of the optimal, respectively. The smaller gap relative to NLP classification stems from two elements. First, accuracy here is measured at the sequence level, granting more flexibility for exiting decisions at individual tokens. Further, auto-regressive generation (with shared state across tokens) grants more continuity than between requests in NLP classification, boosting adaptation benefits.

4.4 Comparison with Existing EE Strategies

Classification. We compare Apparate with two off-the-shelf EE models: BranchyNet [69] and DeeBERT [76]. BranchyNet extends ResNet models with ramps of the same style as Apparate, while DeeBERT extends BERT-base with deeper ramps (using the entire BERT pooler - §3.1). For each, we follow their prescribed architectures, with always-on ramps after every layer. We perform one-time tuning of thresholds as recommended by both works, and consider two variants: the default recommendation where all ramps use the same threshold, and a version that removes this restriction (+).

Table 2 presents our results. The main takeaway is that existing EE approaches, even when favorably tuned, yield unacceptable drops in average accuracy up to 23.9% and 17.8% for CV and NLP. In contrast, Apparate consistently meets the imposed accuracy constraint (1% in this experiment) for both workloads. Further, even with such accuracy violations, tail latencies are 0.9–9.4% lower with Apparate than these systems. The reason is lack of adaptation: all ramps are always active despite current efficacy, yielding undue overheads for non-exiting inputs. In contrast, throughout these experiments, despite having a full ramp budget (for fair comparison), Apparate maintained only 9.1–27.2% of all possible ramps.

For fair median latency comparison, we consider an optimally-tuned (opt) version of existing EE models that perform one-time tuning on the actual test dataset, picking the best (latency-wise) thresholds that ensure <1% accuracy drop. As shown, due to its regular and less-constrained adaptation, Apparate outperforms even this oracle version of existing EEs with up to 14.1% higher median latency savings.

Generative. We compare Apparate with FREE [14], a state-of-the-art EE solution for T5-large that succeeded

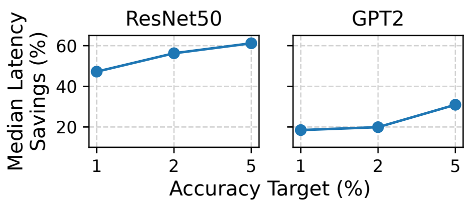


Figure 19. Apparate's wins for different accuracy constraints.

Ramp Budget	ResNet50	GPT2
2%	48.9%	18.5%
5%	49.6%	22.2%
10%	50.4%	24.9%

Table 3. Apparate's median latency wins vs. ramp budget.

CALM [60]. FREE relies on a single, fixed ramp and fine-tunes the entire model to cater to that ramp. The ramp location and weights are selected once to maximize latency savings subject to a 1% accuracy constraint on a representative dataset (default to the first 3% of samples). Both FREE and Apparate use T5's final decode head as the ramp architecture. Moreover, to manage tail latencies, Apparate (like FREE) uses a ramp budget of 1. The main difference is that Apparate dynamically adjusts the ramp position (and threshold) to maximize latency savings and preserve accuracy; FREE's omission of runtime adaptation yields 5.5% accuracy losses in these experiments.

Despite FREE's accuracy violations, Apparate brings median and 25th percentile TPT savings of 28.0–71.0% and 15.7–28.0% over FREE (Figure 18). 95th percentile latencies with Apparate are 1.8–2.4% lower than with FREE because Apparate regularly flushes a batch decoding once the ramp accumulates a pre-specified number of exited tokens.

4.5 Microbenchmarks

Results here use ResNet50 and GPT2-medium running on a random video and Amazon reviews for classification. Reported trends hold for all our workloads.

Parameter sensitivity. Recall that Apparate ingests values for two key parameters: ramp aggression/budget and accuracy constraint. Figure 19 and Table 3 studies the effect that these parameters have on Apparate's latency wins. The findings are intuitive: Apparate's latency savings over vanilla models decrease as ramp budgets shrink or accuracy constraints tighten. Both trends are a result of Apparate being granted less flexibility for adaptation. Importantly, accuracy constraint has a larger impact on Apparate's wins. The reason is that inter-ramp dependencies result in overlap in the set of inputs that can exit at any ramp when run in isolation; thus, wins from using more ramps eventually hits diminishing returns.

Ramp architectures. Although Apparate opts for using many lightweight ramps, it's adaptation algorithms can support any ramp architecture. To illustrate this, we ran Apparate with DeeBERT's more expensive ramps (§4.4). Overall, we find that these costlier ramps dampen Apparate's latency savings by 4% since they constrain Apparate's runtime adaptation in terms of feasible configurations, i.e., fewer active ramps at

System\Workload	ResNet50	GPT2
Clockwork	(20.2, 37.8)	(689.2, 779.4)
TF-Serve	(24.5, 37.8)	(709.3, 793.1)

Table 4. Apparate on different serving platforms. Results show (median, p95) latency over vanilla models in ms.

any time. Crucially, we note that accuracy constraints were still entirely met due to Apparate's frequent threshold tuning.

Impact of serving platform. Apparate runs atop existing serving platforms, responding to serving and exiting patterns rather than altering platform decisions, e.g., queue management. Table 4 shows that, despite platform discrepancies, Apparate's performance wins are largely insensitive to the underlying platform when CV or NLP workloads are configured with the same SLO goal. For example, median latency savings for the Amazon workload and GPT-2 are within 2.9% when using Clockwork or TF-Serving.

Profiling Apparate. Figure 10 analyzes the run time and optimality of Apparate's threshold tuning algorithm. Beyond that, Apparate includes two other overheads while running: ramp adjustment and communication between its CPU controller and GPUs for runtime monitoring. Ramp adjustment rounds take an average of 0.5 ms. Coordination overheads are also low (and non-blocking for serving - §3) because of Apparate's small ramp sizes (definitions and weights consume ~10KB) and profiling data (simply a top-predicted result with an error score, collectively ~1KB). In total, CPU-GPU coordination delays take an average of 0.5ms per communication, 0.4ms of which comes from fixed PCIe latencies in our setup.

Importance of Apparate's techniques. Apparate's runtime adaptation considers frequent (accuracy-guided) threshold tuning, with periodic ramp adjustments. Table 2 shows the importance of threshold tuning on average accuracies. We also evaluate the importance of ramp adjustment on Apparate's latency wins by comparing versions with and without it. Overall, disabling ramp adjustment results in 20.8–33.4% lower median latency wins, though worst-case latency (and throughput) and accuracy constraints remain continually met.

5 Additional Related Work

Early exit networks. Existing proposals focus on exit *ramp architecture* and *exit strategy* [14, 35, 48, 60, 61, 69, 76, 77, 84] for specific models. Ramp architectures are domain-specific, but replicating the last (few) layers is the common practice [76, 77]; Apparate builds on this and prefers shallow ramps (§3). Existing exit strategies consider *confidence* of the labels [48], *entropy* of the prediction [76], or more sophisticated elements such as counters across ramps [84]. Apparate is agnostic to EE technique, and instead focuses on bringing EEs to arbitrary models and providing runtime management to maximize latency savings and accuracy preservation.

Model optimizations. Much work has focused on creating *variants* of ML models to optimize serving. Some employ compiler techniques to analyze (and optimize) execution at a graph or operator level for lower latency [3, 5]. Other techniques compress models into versions with fewer layers or less weight specificity while adhering to accuracy objectives, e.g., quantization [27, 43, 47], distillation [37, 59], and model pruning [26, 49]. These efforts are complementary to Apparate, which would bring input-level compute adaptation to their outputs, i.e., on optimized or compressed models. Moreover, unlike many of these techniques, Apparate does not alter the original model's weights and instead preserves its full predictive power for hard inputs and constant feedback.

Inference optimizations. Recent work optimizes model serving objectives based on workload characteristics [18, 19, 29, 58, 65, 82, 83]. For instance, Inferline [19] optimizes serving cost while adhering to strict latency constraints using intelligent provisioning and management. Shepherd [83] maximizes goodput and resource utilization by leveraging cross-workload predictability. Despite their impressive results, these works optimize their metric of choice at the expense of latency and do not resolve the latency-throughput tension, which is the focus of our (complementary) work. Mystify [31] and INFaaS [58] generate and choose model variants based on their intent and constraints (including performance). As noted above, Apparate operates on the output of such tools, bringing latency wins to compressed models (§4.2).

Dynamic neural networks. Other techniques boost inference efficiency by adapting model execution [22, 25, 36, 72, 78] at different granularities, e.g., sample-wise, spatial-wise, and temporal-wise. As with exiting, their modulations to serving risk accuracy violations and foregone latency wins with dynamic workloads. We hope that Apparate can motivate and guide adaptation systems to navigate accuracy-latency-throughput tradeoffs for such techniques, but leave that to future work. Other low-level (e.g., GPU kernel level) techniques that optimize execution of dynamic neural networks [21, 81] can benefit Apparate and improve its performance.

6 Conclusion

We present Apparate, the first system that automatically injects and manages early exiting for ML inference. Key to Apparate's ability to alleviate latency-throughput tensions in serving is its use of exiting only for fast results (not compute savings). This provides continual feedback on exits, and powers Apparate's novel adaptation strategies for EE ramps and thresholds. Apparate lowers median latencies by 40.5–91.5% and 10.0–24.2% for diverse CV and NLP workloads, and reduces median time-per-token latencies by 22.6–77.9% for generative workloads, all while meeting accuracy constraints and preserving platform throughputs.

Acknowledgements. We thank Wyatt Lloyd, the SOSP reviewers, and our shepherd, Rong Chen, for their helpful and constructive comments. This work was supported by a Sloan Research Fellowship and NSF CNS grants 2147909, 2151630, 2140552, 2153449, and 2152313.

References

[1] 2013. Web data: Amazon reviews. https://snap.stanford.edu/data/web-Amazon.html.

[2] 2018. Neural Network Exchange Format (NNEF). https://www.khronos.org/nnef/.

[3] 2023. Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators. https://tvm.apache.org/.

[4] 2023. How Microsoft's bet on Azure unlocked an AI revolution. https://news.microsoft.com/source/features/ai/how-microsofts-bet-on-azure-unlocked-an-ai-revolution/.

[5] 2023. NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.

[6] 2024. HuggingFace Pipelines. https://huggingface.co/docs/transformers/en/main_classes/pipelines.

[7] 2024. NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

[8] 2024. ONNX Run Time. https://github.com/microsoft/onnxruntime.

[9] 2024. Open Neural Network Exchange (ONNX). https://onnx.ai/.

[10] 2024. The Yelp Reviews Dataset. https://www.yelp.com/dataset.

[11] 2024. TorchServe. https://pytorch.org/serve/.

[12] Neil Agarwal and Ravi Netravali. 2023. Boggart: Towards General-Purpose Acceleration of Retrospective Video Analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 933–951. https://www.usenix.org/conference/nsdi23/presentation/agarwal-neil

[13] Shimaa Ahmed, Amrita Roy Chowdhury, Kassem Fawaz, and Parmesh Ramanathan. 2020. Preech: A System for Privacy-Preserving Speech Transcription. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2703–2720. https://www.usenix.org/conference/usenixsecurity20/presentation/ahmed-shimaa

[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] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 119–135. https://www.usenix.org/conference/nsdi22/presentation/bhardwaj

[16] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. 2017. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*. PMLR, 527–536.

[17] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. 2019. Scaling video analytics on constrained edge nodes. *Proceedings of Machine Learning and Systems* 1 (2019), 406–417.

[18] Y. Choi, Y. Kim, and M. Rhu. 2021. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 493–506. https://doi.org/10.1109/HPCA51647.2021.00049

[19] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[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] Harm de Vries, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron Courville. 2017. Modulating early visual processing by language. arXiv:1707.00683 [cs.CV]

[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] Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, et al. 2024. Layer skip: Enabling early exit inference and self-speculative decoding. *arXiv preprint arXiv:2404.16710* (2024).

[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] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. arXiv:2301.00774 [cs.LG]

[27] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG]

[28] Gigaspaces. 2023. Amazon Found Every 100ms of Latency Cost them 1% in Sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales.

[29] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/3135974.3135993

[30] 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

[31] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 705–719. https://www.usenix.org/conference/nsdi21/presentation/guo

[32] 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.

[33] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597

[34] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 269–286.

[35] 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]

[36] 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]

[37] Yukun Huang, Yanda Chen, Zhou Yu, and Kathleen McKeown. 2022. In-context Learning Distillation: Transferring Few-shot Learning Ability of Pre-trained Language Models. arXiv:2212.10670 [cs.CL]

[38] HuggingFace. 2023. Pretrained Models. https://huggingface.co/transformers/v3.3.1/pretrained_models.html.

[39] Anand Iyer, Mingyu Guan, Yinwei Dai, Rui Pan, Swapnil Gandhi, and Ravi Netravali. 2024. Improving DNN Inference Throughput Using Practical, Per-Input Compute Adaptation. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. https://doi.org/10.1145/3694715.3695978

[40] 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

[41] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *International conference on machine learning*. PMLR, 3301–3310.

[42] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. 2023. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 917–932. https://www.usenix.org/conference/nsdi23/presentation/khani

[43] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. 2024. SqueezeLLM: Dense-and-Sparse Quantization. arXiv:2306.07629 [cs.CL]

[44] 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*. 611–626.

[45] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. arXiv:2211.17192 [cs.LG]

[46] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[47] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 [cs.CL]

[48] 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

[49] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. arXiv:2305.11627 [cs.CL]

[50] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16. https://doi.org/10.1109/MM.2020.2974843

[51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139 [cs.DC]

[52] Arthi Padmanabhan, Neil Agarwal, Anand P. Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. 2022. GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. *CoRR* abs/2201.07705 (2022). arXiv:2201.07705 https://arxiv.org/abs/2201.07705

[53] Aditya Pal, Abhilash Barigidad, and Abhijit Mustafi. 2020. IMDb Movie Reviews Dataset. https://doi.org/10.21227/zm1y-b270

[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] PyTorch. 2023. Model Zoo. https://pytorch.org/serve/model_zoo.html.

[56] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[57] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[58] 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)*. 397–411.

[59] 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).

[60] 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.

[61] 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

[62] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

[63] Bart Selman and Carla P Gomes. 2006. Hill-climbing search. *Encyclopedia of cognitive science* 81 (2006), 82.

[64] Jaime Sevilla, Pablo Villalobos, and Juan Cerón. 2021. Parameter counts in Machine Learning. https://www.lesswrong.com/posts/Gz

oWcYibWYwJva8aL/parameter-counts-in-machine-learning.

[65] 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

[66] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 821–839. https://www.usenix.org/conference/osdi22/presentation/sima

[67] Abhijit Suprem, Joy Arulraj, Calton Pu, and Joao Ferreira. 2020. ODIN: Automated Drift Detection and Recovery in Video Analytics. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2453–2465. https://doi.org/10.14778/3407790.3407837

[68] Sudeep Tanwar, Sudhanshu Tyagi, Ishan Budhiraja, and Neeraj Kumar. 2019. Tactile Internet for Autonomous Vehicles: Latency and Reliability Analysis. *IEEE Wireless Communications* 26, 4 (2019), 66–72. https://doi.org/10.1109/MWC.2019.1800553

[69] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. arXiv:1709.01686 [cs.NE]

[70] Think with Google. 2017. The Need for Mobile Speed: How Mobile Latency Impacts Publisher Revenue. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-speed-latency-impacts-publisher-revenue/.

[71] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[72] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. SkipNet: Learning Dynamic Routing in Convolutional Networks. arXiv:1711.09485 [cs.CV]

[73] 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.

[74] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344. https://doi.org/10.1109/HPCA.2019.00048

[75] 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

[76] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. arXiv:2004.12993 [cs.CL]

[77] 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

[78] Yu-Syuan Xu, Tsu-Jui Fu, Hsuan-Kung Yang, and Chun-Yi Lee. 2018. Dynamic Video Segmentation Network. arXiv:1804.00931 [cs.CV]

[79] Shuo yiin Chang, Bo Li, David Johannes Rybach, Wei Li, Yanzhang (Ryan) He, Tara N Sainath, and Trevor Deatrick Strohman. 2020. Low Latency Speech Recognition using End-to-End Prefetching. In *Interspeech 2020*.

[80] 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

[81] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. 2023. Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 681–699.

[82] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1049–1062. https://www.usenix.org/conference/atc19/presentation/zhang-chengliang

[83] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[84] 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

7 Appendix

The appendix was not peer-reviewed.

We list the SLOs for each model in the classification experiments (Figures 12–14). The default SLOs for classification models are 2× each model's inference time with batch size 1.

Model	Latency w/ bs=1 (ms)	Default SLO (ms)
ResNet18	6.5	13.0
ResNet50	16.4	32.8
ResNet101	33.3	66.6
VGG11	3.3	10.0
VGG13	3.8	10.0
VGG16	4.5	10.0
Distilbert-base	15.5	31.0
BERT-Base	29.4	58.8
BERT-Large	63.2	126.4
GPT2-medium	103.0	206.0

Table 5. Different SLOs used in §4.2 and Figure 17. All numbers are measured on the A6000.

We also list the pseudocode for our threshold tuning and ramp adjustment algorithms described in the design section.

Algorithm 1: Threshold tuning algorithm

```
Input: ramps, list of ordered active ramp IDs
Input: acc_loss_budget, max accuracy loss tolerable
Input: smallest_step_size, smallest step size for
       incrementing thresholds
Output: thresholds, thresholds associated with each ramp
Output: latency_savings, latency savings with searched
        thresholds
/* all thresholds start at 0, i.e. no EE */
1  $thresholds \leftarrow [0.0] * len(ramps)$
/* each ramp has its own step size */
2  $step\_sizes \leftarrow [smallest\_step\_size] * len(ramps)$
3  while True do
4      $best\_ramp, overstepped\_ramps, latency\_savings \leftarrow$
           $pick\_ramp(ramps, thresholds, acc\_loss\_budget)$
       /* find next ramp to update thresholds */
5      if best_ramp is valid then
           /* increment threshold of the selected ramp */
6          $thresholds[best\_ramp] += step\_sizes[best\_ramp]$
           /* double step_size */
7          $step\_sizes[best\_ramp] *= 2$
8      else
9          if $step\_sizes.all() \leq smallest\_step\_size$ then
10             return thresholds, latency_savings
       /* half step_size for overstepped ramps,
          double step_size for the rest */
11     for ramp in overstepped ramps do
12         $step\_sizes[ramp] /= 2$
```

Algorithm 2: Ramp adjustment algorithm

```
Input: P, latest positive ramp
Input: ramps, list of ramp ids
/* Initialize ramp settings */
/* Check whether negative ramps exist */
1  if negative utility ramps exist then
2      Try fast threshold tuning
3      if still negative then
4          Deactivate all negative-utility ramps
5      else
6          Update thresholds
7          return
       /* Determine candidate ramps located at
          the middle of each interval after P
          (see Figure 11) */
8      $candidates \leftarrow pick\_candidates(P, ramp\_ids)$
       /* Compute utilities and select ramp with
          best upper-bound utility (see
          Figure 11) */
9      foreach candidate in candidates do
10         Compute upper-bound exit rate
11         Calculate utility using upper-bound exit rate
12         if utility is positive and utility >
             selectedRamp.utility then
13             $selectedRamp \leftarrow candidate$
       /* Update EE configuration */
14     Remove all deactivated ramps
15     Add selectedRamp with initial threshold=0
16     Update thresholds in future tuning rounds
/* If all ramps have positive utilities,
   explore the possibility of higher latency
   savings */
17 if all utilities positive then
18     if ramp budget allows then
19         Add a ramp before the highest utility ramp
20     else
21         Shift the lowest utility ramp one position earlier
22 return
```