Tabi: An Efficient Multi-Level Inference System for Large Language Models

Yiding Wang[1], Kai Chen[1], Haisheng Tan[2], Kun Guo[3]
[1]iSING Lab, Hong Kong University of Science and Technology
[2]University of Science and Technology of China [3]Fuzhou University

Abstract

Today's trend of building ever larger language models (LLMs), while pushing the performance of natural language processing, adds significant latency to the inference stage. We observe that due to the diminishing returns of adding parameters to LLMs, a smaller model could make the same prediction as a costly LLM for a majority of queries. Based on this observation, we design Tabi, an inference system with a multi-level inference engine that serves queries using small models and optional LLMs for demanding applications. Tabi is optimized for discriminative models (i.e., not generative LLMs) in a serving framework. Tabi uses the calibrated confidence score to decide whether to return the accurate results of small models extremely fast or re-route them to LLMs. For re-routed queries, it uses attention-based word pruning and weighted ensemble techniques to offset the system overhead and accuracy loss. We implement and evaluate Tabi with multiple tasks and models. Our result shows that Tabi achieves 21%-40% average latency reduction (with comparable tail latency) over the state-of-the-art while meeting LLM-grade high accuracy targets.

1 Introduction

Transformer-based language models, such as BERT [18], have achieved strong performance in various natural language processing (NLP) tasks. One key to their advancement is the sheer volume of parameters and we are seeing a trend of building ever larger language models (LLMs) to push the limit of NLP performance [9, 34]. LLMs are computationally intensive, resulting in high inference latency [80, 89], whereas low latency is crucial to the user experience of NLP applications.

Given the high computational demands of LLMs, it is common to use the cloud infrastructure to deliver their breakthroughs [71]. Machine learning (ML) inference systems [15, 31, 57] serve incoming queries of various applications with trained deep neural networks (DNNs) on the server. They select appropriate models for the user to meet tasks' accuracy and latency requirements so users do not need to, letting users deploy trained models efficiently without mastering ML or system optimizations (i.e., *model-less* inference [57]).

However, the current inference systems mostly overlook the concerning status quo of NLP: *LLMs trade a huge amount of DNN capacity for top-grade accuracy, while the return diminishes* [11]. Studies find that the pace of model growth far exceeds the resulting increase in model performance and more resources are required to improve language models by making them larger [61]. Model compression [28, 59, 75] has attracted considerable research attention, but the consequent accuracy loss prevents compressed models from serving accuracy-demanding tasks. Adaptive inference techniques (e.g., early-exit[80, 89] and token pruning [29, 74]) require special DNN designing and training, thus are hard to fit in with today's model-less inference systems.

When serving accuracy-demanding applications that do require LLMs, the current inference systems select the same models to serve all queries of a task homogeneously [31, 57]. They ignore the variance of difficulty between individual data samples within an NLP application [10]. This is because there is no easy way to set per-query requirements in advance. Essentially, inference systems treat each DNN query as a black-box[1] without understanding its specific needs and focus more on resource scheduling instead.

We argue that optimizing for all queries homogeneously when serving NLP applications is sub-optimal because the LLMs selected could be a huge overkill for most less demanding data samples. For example, we find that to serve a text classification task with a strict accuracy target, INFaaS [57] needs to select a highly accurate but heavy LLM. Meanwhile, a faster model with only 1/4 of parameters can correctly fulfill over 90% of the queries. Yet it will not get selected for its below-target accuracy. Such huge resource overheads following today's trend of developing larger models motivate us to revisit the coarse-grained model selection when serving LLMs and optimize single data inference.

We propose Tabi, an inference system featuring a novel *multi-level inference engine* driven by individual per-query feedback and employing recent advances of ML to optimize the inference latency of LLMs for discriminative tasks. Unlike today's inference systems that serve the whole application with the same models, Tabi incorporates multiple DNNs each optimized for efficiency or accuracy to handle heterogeneous queries within a task and avoid invoking the costly LLMs when smaller models can suffice.

[1]The previous white-box DNN inference engine [45] focuses on a different aspect of reusing low-level operations to improve computational efficiency.

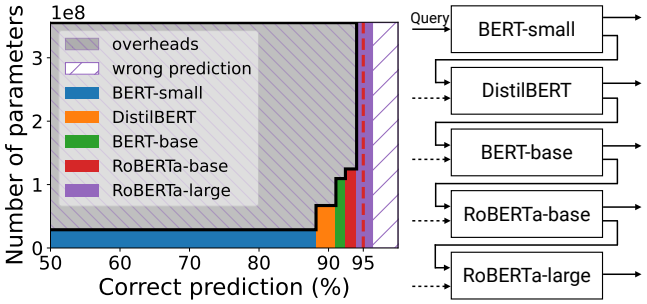
In Tabi, each query will first go through a level of efficient DNN, obtaining predictions and a confidence score well-calibrated with temperature scaling [32] that accurately indicates the inference quality. We use a probability-based dispatcher to decide a query's DNN capacity: For a majority of highly confident predictions, we directly return the results and achieve extremely low inference latency; if not sure, Tabi re-routes those challenging queries to the next level of a more advanced model for high-quality inference (§4.1).

Tabi employs ML techniques and reuses results of the small model to optimize the system overhead of occasionally running extra DNNs. For re-routed queries, we directly prune the words that contribute little to the task from the input data using the previous DNN's *attention weights* [69], offsetting the small model's overhead by reducing the LLM's latency instead (§4.2). Attention weights are intermediate variables of language models. An encoded token is semantically important if it receives a lot of attention from other tokens [79]. Since natural language is highly redundant (due to less meaningful elements such as prepositions and punctuation marks), removing low-attention words can accelerate inference while keeping accuracy [29, 74]. When outputting the results, Tabi further improves the LLM's accuracy by combining existing predictions using *weighted ensemble* without requiring extra DNN computations (§4.3). Through designs inspired by recent advances of ML/NLP, Tabi achieves fine-grained improvements over state-of-the-art (SOTA) inference systems.

We abstract a set of Tabi model levels and configurations as a *model candidate*, which is logically equivalent to single models and compatible with inference scaling systems (e.g., INFaaS). Meanwhile, we focus on single data inference (i.e., each query runs separately), which is fundamental to live data analytics and can benefit complex workloads [73]. To navigate the large optimization space brought about by the general system architecture, Tabi conducts efficient offline profiling to generate differently optimized candidates and selects the optimal candidate for incoming tasks (§5).

We evaluate Tabi using multiple NLP benchmarks and popular language models. Tabi achieves a 21%-40% reduction on average latency and good tail performance compared to SOTA inference systems while meeting the LLM-grade demanding accuracy targets. Tabi can work together with ML-optimized models (e.g., via compression) by using them in candidates and can switch to them for relaxed accuracy targets. Compared to ML research on adaptive inference (e.g., early-exit[80, 89] and token pruning [29, 74]), Tabi can accelerate various generic language models for model-less inference and achieve even better performance without requiring customization and extra training for every model.

To summarize, the contributions of Tabi include: (1) proposing the first inference system for the resource overhead issue among increasingly large language models; (2) designing an efficient multi-level system empowered by ML advancements that achieves significant latency reduction while meeting demanding accuracy targets; and (3) optimizing for generality across models without requiring customization.



(a) Resource overheads of LLMs. **(b)** An *imaginary* MLFQ.

Figure 1. Each color-filled bar's right edge shows its accuracy. A bar's width shows the accuracy improvement over the previous smaller DNNs, i.e., the percentage of queries that can be correctly served by a model but not by the ones on its left. The height shows the model size. The gray area is the resource overheads compared to an ideal scenario.

2 Background and Motivation

2.1 Resource Overhead of LLMs

We are in the paradigm of large language models (LLMs) today. BERT [18], published in 2018, impressively improved the GLUE (General Language Understanding Evaluation) benchmark consisting of 9 tasks [72] from no higher than 69 out of 100 to 80. Since then, masked language modeling [18] proposed by BERT and other self-supervised pre-training objectives enable the utilization of large-scale datasets consisting of billions of words.

Learning from massive amounts of data to substantially understand the language requires large-scale models. For example, BERT-large with 340 million parameters was one of the largest DNNs when proposed. Switch Transformer [24] uses more than a trillion parameters to achieve SOTA results on several cognitive tasks. In recent years, LLMs with the attention mechanism [69] and similar architectures as BERT have passed the human baseline and pushed the GLUE leaderboard above 90.

On the other hand, achieving SOTA performance requires more and more computational resources while the return is diminishing [61]. For example, a SOTA Transformer model spends 50% of its training footprint solely to achieve a decrease of 0.3 of the error rate [53]; the recent DeBERTa [35] uses 36× more parameters than ALBERT [43] but only improves the accuracy by 6%. GPT-3 [13], which has 175 billion parameters, 100× of its predecessor, suggests that the current LLMs "may eventually run into the limits of the pre-training objective".

Figure 1a shows the accuracies and sizes of five DNNs from small to large for text classification. We can see that the accuracy improvements over previous models (widths of bars) become less noticeable even when adding more parameters (height differences of bars). Our finding echoes sustainable ML research that alarms the vast carbon footprint of training and deploying LLMs [61, 65]. Research also suggests that for less demanding tasks, especially on edge devices, such large DNNs are not necessary when much smaller models might suffice [27, 70].

The NLP status quo of developing larger models will cause huge resource overheads when delivering their SOTA performance using the current DNN inference systems. To understand the issue, let us look at how they serve a text classification task (SST-2 [72]) with a target accuracy of 95% in Figure 1a. To meet the demanding target (vertical dashed line), the inference system chooses RoBERTa-large from available trained models, which has an accuracy of 96.3% and 355 million parameters. The workload's resource usage is generally equivalent to the area of the whole figure, serving all queries (width) with the selected DNN's capacity (height).

However, when analyzing the results *post hoc*, we find that other more efficient models can make the exact prediction as RoBERTa-large does for over 90% of queries (e.g., DistilBERT, the second smallest model in Figure 1a, has an accuracy of 91%), although none of them meet the accuracy target to get selected (see how model selection works in §2.2).

Ideally speaking, if a query can be served correctly by a small model, we should not invoke larger ones.[2] This scenario is similar to going through a multi-level feedback queue (MLFQ) of models from small to large as Figure 1b while *knowing* the shortcut (the dashed arrows) to the right level. Thus, the optimal resource usage of a workload is the lower right area of all bars in Figure 1a, which is only 11% of the actual usage! We define easy queries as those obtaining the same results no matter served with small or large DNNs. While the left gray area is the overhead of unnecessarily serving easy queries with over-powerful LLMs.

In reality, we do not know a query's minimum requirements *beforehand*, and thus there is no shortcut to the most efficient model; besides, going through multiple models adds extra latency. This is why the current inference systems match performance targets at an application level and focus on resource/model scaling instead. Nevertheless, the remarkable potential of latency reduction, cost saving, and environmental impact motivates us to optimize the inference of each single query for LLMs in a model-less system.

2.2 DNN Inference Systems

Recent inference systems [31, 57] let users only specify high-level performance requirements (e.g., accuracy and latency for a sentiment analysis task) rather than specific models and deploy trained models without mastering ML or system optimizations, termed as *model-less* inference. The system first selects appropriate resources (e.g., GPU or CPU) and DNNs (e.g., optimized for accuracy or efficiency) from registered models. Then it routes queries to the selected models, runs DNN inference, and returns the results. This paper focuses on model selection and single data inference. In addition, inference systems can also scale the resources and models to adapt to workload changes.

The current inference systems make various optimizations. INFaaS [57] selects the best single models automatically optimized for different devices and batch sizes for inference and quickly adapts to workload changes. Cocktail [31] employs ensemble learning with multiple small models to reduce latency through parallel execution; it also dynamically adjusts model ensembles to minimize cost.

SOTA inference systems set performance targets for an application and select models to serve the whole workload *homogeneously* without inspecting whether the selected models are overkill for individual queries. Although Cocktail [31] has a monitoring process using ground-truth labels which are rare for real-world tasks, it cannot achieve fine-grained model selection. This one-model-fits-all design is not a concerning problem for many tasks when the DNN performance improves notably as the model grows larger (i.e., less per-query overheads), which is not the case for LLMs.

We observe an apparent variance of needs for DNN capacity within an NLP application: The natural workload is a mix of difficult and easy samples and many easy samples can be served correctly by a much smaller language model. ML research has been exploring the variance in training and inference settings: Hard sample mining [37, 63] spends more training resources on complex data, and adaptive prediction [10, 68, 80] develops specialized DNNs that only run early layers for easy data. Tabi tackles the overhead issue of serving LLMs from a model-less inference perspective.

2.3 Pay Attention to Transformers

Tabi focuses on serving *discriminative* LLMs including BERT and 25 of the 30 most downloaded models [2]. Such models summarize the inputs, learn unique proprieties, and make predictions, which are different from *generative* models that generate new tokens like GPT-3 [13] and machine translation [69]. Such pre-trained language models power various NLP tasks that feed on encoded text representation, including non-Latin [16], programming code [25], and clinical [36] languages, such as sentiment analysis [48], natural language inference [72], question answering [54], and applications including review understanding [81] and content moderation [58]. Tabi does not optimize for generative models whose Transformer decoder architecture does not support our confidence score and word pruning design.

[2] As the Occam's razor principle goes, *entities should not be multiplied beyond necessity.*

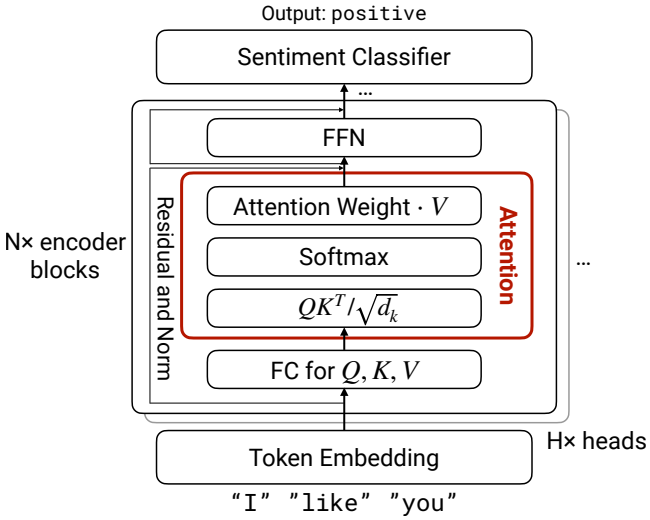


Figure 2. Illustration of the attention module in BERT, which is similar to those in other language models.

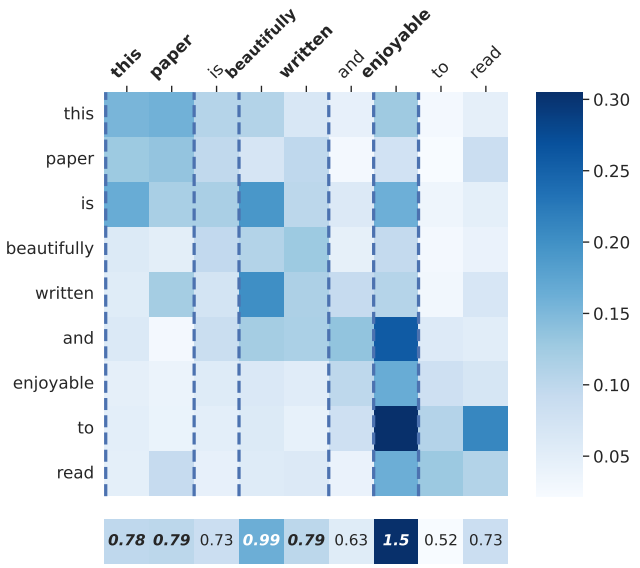


Figure 3. The attention weight matrix and the importance vector. Tokens with top-50% weights are in **bold**. They are more important to the task. Special tokens are omitted.

Language model architecture. Discriminative language models consist of a series of *Transformer encoder blocks*. We first turn words into a numeric representation. Language models split an input sentence into sub-word tokens and map them to numeric vectors before feeding them to the Transformer blocks. This step is called *tokenization*. A Transformer block is a particular neural network: As shown in Figure 2, the input vector will go through a self-attention module [69] connected to a feed-forward network (FFN). The inputs of attention consist of Query (Q), Key (K), and Value (V) with the same dimension (d_k), obtained by multiplying the input vector by their weights, each split into multiple heads. QKV are transformed into the intermediate attention output. Then, an FFN will apply to the attention output and produce input for the next Transformer block. This process will repeat as going through the hidden layers of LLMs.

Attention mechanism. Language models build on the attention mechanism. Conceptually, attention encodes the relationship between tokens, generates context-dependent embeddings, and lets models know the relevant tokens when processing a vector. First, we calculate $QK^T/\sqrt{d_k}$ to produce the attention score, which shows how close two tokens are related. Intuitively, attention works like searching and retrieving in a database, which is why it uses scaled dot-product to measure the similarity between tokens represented by Q and K. Then we apply softmax to obtain the *attention weight* of a query to enlarge the score for highly-related token pairs. Multiplying the attention weights with V gives the result of one head. To summarize, the attention mechanism works as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

The multi-headed mechanism concatenates the output of each attention head and lets the model focus on different positions in parallel [51].

Attention weights. The attention weight $softmax(\frac{QK^T}{\sqrt{d_k}})$ normalizes the "attention" paid from one token to all the other tokens in an input sequence to a probability distribution that sums to 1 (rows in Figure 3). Thus summing up the attention weights received from all tokens will produce an importance vector (bottom of Figure 3), in which *more impactful tokens that elicit more attention have higher weights*.

For example, Figure 3 shows the attention weight matrix of an input sentence produced by a sentiment analysis BERT model. Tokens that express strong emotion (e.g., "beautifully" and "enjoyable") have the highest weights in the importance vector, while adpositions (e.g., "and" and "to") receive the least attention. Nevertheless, all tokens, whether important or not, contribute to the computational complexity of attention quadratically regarding the sequence length [69].

ML research finds that dropping unimportant vectors across layers can reduce computation cost [29, 74] but they require specially designed and trained models. Tabi accelerates generic LLMs without code modification or re-training by directly reducing the input data size (rather than per-layer pruning) using the attention weights available in the system.

3 Tabi Overview

Motivated by the huge overheads of serving LLMs with today's model-less inference systems and new optimization opportunities brought about by the attention mechanism,

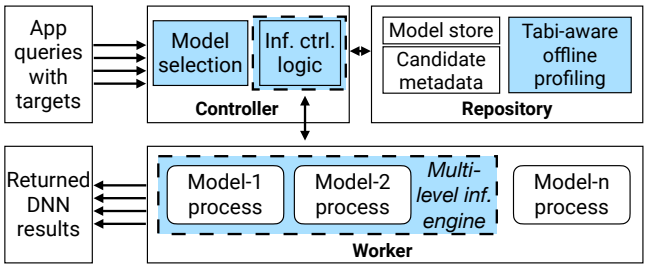


Figure 4. Tabi workflow. Highlighted components are optimized in this work. Components in dashed lines form the logical *multi-level inference engine* (§4).

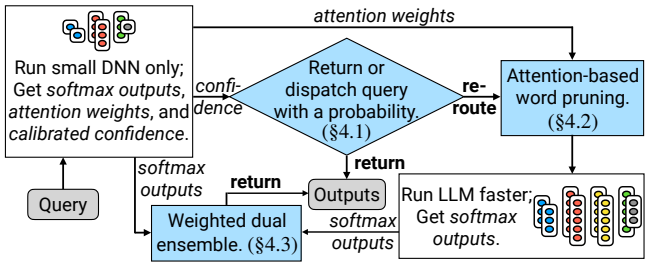


Figure 5. Overview of multi-level inference engine with two model levels. 3 system modules are highlighted. Generated variables are in *italic*, and system decisions are in **bold**.

Tabi proposes a novel *multi-level inference engine* to achieve fine-grained latency reduction efficiently for generic models driven by real-time per-query feedback.

Figure 4 illustrates the workflow and components of Tabi. The model repository stores the trained DNNs submitted and maintained by the user. Tabi conducts offline profiling to obtain the statistics of single models and various multi-level DNN configurations (i.e., *candidates*, details in §5). For a new task with performance targets, the controller selects the optimal candidate and runs DNN inference on the worker. The inference control logic consists of the three modules in §4 and manages the DNNs used on the worker. We make contributions in model selection and single data inference while the design of Tabi has no substantial difficulty to work with resource/model scaling systems [31, 57].

Model repository. Tabi maintains a model repository that stores the registered DNNs and their metadata, including the accuracy, inference latency, memory utilization, and intermediate results including attention weights and softmax outputs. In addition to single models, we efficiently profile various configurations of Tabi formed by available models and view them as logical model candidates.

Model selection. As discussed in §2.1, the theoretically ideal solution ignores the prohibitively large cost of enumerating *all* DNNs to find the perfectly fitting one for a single query. To handle this in practice, Tabi generates differently optimized candidates to bound

the overhead and balance the performance. For queries of a specific task, Tabi selects the optimal model candidate regarding the the target and runs inference. We further elaborate the criterion in §5.

4 Multi-Level Inference Engine

The core of Tabi is a novel *multi-level inference engine* that employs both efficient and highly accurate DNNs to serve heterogeneous NLP queries with corresponding resources, as shown in Figure 5. In addition to the DNNs, the inference engine includes (1) a probability-based dispatcher that uses the *calibrated confidence score* to return accurate predictions fast; (2) input data pruning using existing attention weights to accelerate LLM inference for re-routed queries and reduce system overheads; and (3) a weighted multi-level ensemble of activated models to further increase the final accuracy.

Compared to ML-native optimizations. Tabi conceptually echoes the ML research on adaptive inference (e.g., early-exit[80, 89] and token pruning [29, 74]) in terms of consuming only necessary DNN resources. We find that letting a DNN decide which layer to return results or which tokens to drop requires non-trivial and, more importantly, per-model non-scalable labor, including modification to the model architecture and additional training efforts. This is not ideal because the booming of *pre-trained* language models and *model-less* inference systems were meant to save efforts for both ML development and deployment. Instead, Tabi achieves low inference latency with DNNs straight from model zoos without customization. Tabi implements the attention-aware multi-level design at the system level inspired by ML advancements, so that we can directly serve generic or generated models (e.g., via compression [28]) with optimized performance and usability of model-less inference. We evaluate Tabi compared to early-exit and optimized single models in §7.5.

4.1 Probabilistic Dispatcher

In applications that handle complex natural languages, DNNs should not only be accurate but also need to indicate when they are likely to be incorrect. In Tabi, returning the incorrect results of small models early could nullify any advantages of LLMs and risk the violation of accuracy targets. We design a probabilistic dispatcher to decide when to return outputs early and when to continue the inference.

A straightforward way to get the confidence of prediction is using the softmax probability, which sums to 1. This method is used by the popular Hugging Face deployment pipeline [22] and its model zoo. The probability of a class i among the set of all classes K is calculated as:

$$P(y_i) = softmax(logits_i) = \frac{\exp(logits_i)}{\sum_{j \in K} \exp(logits_j)} \quad (2)$$

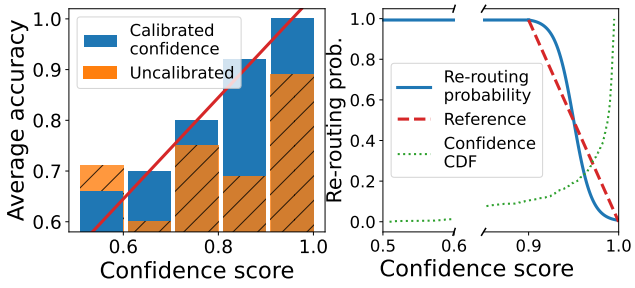


Figure 6. Confidence score buckets and their average accuracy for SST-2 (left). Confidence calibration allows us to make accurate dispatching decisions. Our re-routing probability curve (right) can balance the accuracy and efficiency.

where *logits* is the raw model output.

However, modern DNNs, including language models, are "over-confident" [17, 32]. In our test, small models constantly make predictions with softmax probabilities over 0.9, which are higher than their actual accuracies; besides, even wrong predictions are often made with high softmax probabilities.

In the left part of Figure 6, we group the results of a binary classification workload by confidence score (Equation 2, range from 0.5 to 1) into five buckets and show their average accuracies as the patterned orange bars. The red diagonal line illustrates the ideal relationship between the confidence score and the actual inference accuracy when they are equal. Clearly, they do not match and high softmax probabilities (i.e., above 0.8) do not guarantee high accuracies, making it hard to understand the inference quality.

To dispatch queries reliably and balance accuracy and efficiency, we use temperature scaling [32] to calibrate the softmax confidence score $con(y)$ so that it can match the actual inference accuracy:

$$con(y) = \max_{\forall i \in K} \frac{\exp\left(logits_i/T\right)}{\sum_{j \in K} \exp\left(logits_j/T\right)} \tag{3}$$

where T is the temperature set through a quick fitting.

As a result, the calibrated confidence score (blue bars in Figure 6) matches the ideal distribution of accuracy, providing Tabi reliable basis to decide when to return the correct results of the efficient model level quickly.

Calibrating with Equation 3 does not require DNN retraining. A quick fitting that minimizes the negative log-likelihood loss can set the optimal T. We calibrate the trained models in the model repository together with the offline profiling. We use appropriate techniques such as multi-class calibration [42] and regression calibration [41].

With the calibrated confidence score $con(y)$, we define the probability of re-routing a query to the next level in Equation 4, which is a combination of the normalized confidence, the ReLU, and the scaled sigmoid activation functions:

$$P_{large}(y) = \begin{cases} \dfrac{1}{1+\exp\left(k \cdot norm(con(y))\right)} & con(y) > c \\ 1 & con(y) \le c \end{cases},$$
$$norm(con(y)) = \frac{con(y) - c}{1 - c} - \frac{1}{2} \tag{4}$$

where c is the cut-off threshold close to 1 and k is a scaling parameter for the slope of the sigmoid. Both c and k are task-specific for the various distribution of difficulty variance. Tabi offline-profiles them with the DNN models and will be set at the model selection phase to achieve optimal performance (§5). Equation 4 combines soft probability and hard cut-off to balance accuracy (high recall of challenging queries) and latency (high precision of re-routing). We evaluate how hyperparameters affect different tasks in §7.4.

The right-hand side of Figure 6 illustrates Equation 4 with $c = 0.9$ and two classes. For queries whose confidence is lower than the cut-off threshold, the dispatcher always re-routes them to the next-level model. For the more certain queries, we use the *scaled sigmoid* function to prioritize those with near-100% confidence for returning and adjust the re-routing probability of queries whose confidence is just above c for better accuracy (solid line). Compared to linearly decreasing the re-routing probability (dashed line), our scheme can return more queries early for lower latency and achieves higher accuracy among the returned ones.

Although the probability of returning early (area above the solid line) seems limited, we find that the CDF of queries (green dotted line) usually concentrates around the high-confidence side, so the potential is huge. In our evaluation, the dispatcher can return around 50% to 70% of the queries after the first efficient model. This is because our model selection process can use a competent and efficient model as the first level to share the workload of LLMs.

4.2 Attention-Based Word Pruning

After the efficient level and dispatching, those challenging queries still require LLMs. A concern about serving such queries with extra small models is that re-routed queries would increase the tail latency even though we improve the average and median performance. On the contrary, we find that the small model's results can reduce the latency. We use the real-time semantic understanding (i.e., attention weights) of a query made by the small model to accelerate its LLM inference by pruning unnecessary words from the input data and offset the multi-level latency overhead.

Natural languages contain redundant tokens such as prepositions and postpositions that contribute less to the NLP task [14, 29]. Pruning such words in a query can accelerate inference because the computational complexity of the attention mechanism grows quadratically with the sequence length [69]. Recent ML research develops specialized DNN

structures to adaptively prune tokens across layers based on attention weights [29, 74]. Nevertheless, such optimization requires specialized models and extra training, so it does not fit into model-less inference systems. Instead, we directly optimize the input texts of the LLM at the model level rather than layer level through the attention weights readily provided by the previous small model.

As introduced in §2.3, language models build on the attention mechanism, which indicates the semantic importance of each sub-word token. We obtain the importance vector I (as illustrated in Figure 3) of a query by extracting and accumulating the attention weights of each layer and head of the previous model, as shown in Figure 5. Attention layers and heads handle different aspects of the language [14, 56]; thus, accumulation makes the importance vector more reliable. The attention weight is an intermediate variable produced during inference, so this summation process is fast.

To select the significant tokens and generate a pruning mask, we normalize I to relative z-scores and convert them to 1s and 0s (to be pruned) with a binary step:

$$prune\_mask(I) = \begin{cases} 1 & I > \alpha \cdot \mu \\ 0 & I \leq \alpha \cdot \mu \end{cases} \quad (5)$$

where μ is the mean value of I and α is the hyperparameter of relative pruning degree. We set α through model profiling and selection (§5). We always keep special tokens.

Having the *token* pruning mask is not enough because different language models usually have different numeric representation methods of tokens called *tokenizers*. For example, some tokenizers will split and encode the word "beautifully" into two tokens, "beautiful" and "-ly", while others encode it as a whole. Besides, tokenizers with different vocabularies will also map the same word to different numeric word ids. We find that the only universal data passing interface between models is the text itself.

Our inference system should optimize for generality across models with different architectures and embedding formats. Instead of pruning tokens across layers [29], Tabi generates a *word* pruning mask and applies it on the raw input texts to the LLM, e.g., "this paper beautifully written enjoyabe" is the optimized query from the toy example in Figure 3. We prune a word only if all of its sub-word tokens are 0s in Equation 5, using token's positional information from the tokenizer. For instance, if "beautiful" should be preserved while "-ly" is masked, Tabi keeps the whole word for accuracy.

Attention-based semantic understanding is universal across language models and we experimentally validate the correctness of word pruning using another model's attention weights. We find that smaller models produce similar distributions of importance vectors compared to large ones, tested on SST-2 dataset: 89% of the top-50% important tokens are the same between a small and a large model, indicating Tabi can maintain the necessary tokens. The difference is that LLMs are better at *clearly* distinguishing high- and low-importance tokens. Theoretical research suggests that layers and heads in LLMs work as an ensemble of low-rank attention layers [27, 51]; removing some (i.e., using small models) does not drastically alter the attention outputs [56].

4.3 Weighted Multi-Level Ensemble

For those re-routed queries, rather than directly outputting the prediction of the final-level LLM, we employ *weighted ensemble learning* to combine it with the previous levels' predictions to improve accuracy. This technique does not add extra ML computations since all the intermediate softmax outputs are readily available in the multi-level structure.

An ensemble is a set of models whose individual decisions are combined with weights to make predictions jointly. It has been proved that adding independent models, even weak ones, to an existing large model can be more accurate than a single large model alone because of the reduced variance and bias [40, 49].

When the LLM outputs its results, Tabi forms an weighted average prediction $s(y)$ (i.e., *soft voting*) with existing predictions of the efficient model. For example, a two-level Tabi configuration outputs:

$$s(y) = w \cdot softmax_2(y) + (1 - w) \cdot softmax_1(y) \quad (6)$$

where w is the weight parameter of the final LLM in the ensemble algorithm [12]. We use the standard method to set w at the model selection stage: Weights are proportional to the accuracy on the re-routed data during offline profiling [19].

The recent inference system Cocktail [31] proposes an ensemble of smaller DNNs to achieve parallel execution and reduce latency, which essentially makes LLMs shallower but wider. Different from Cocktail, Tabi uses ensemble to make up for the slight accuracy loss during the previous latency-focused optimizations at almost no additional cost. Latest biomedical research proposes to adaptively set the ensemble weight using each data sample's confidence score [50] which is available in Tabi, but we find the overall improvement limited and decide to keep the system simple.

5 Model Candidate Profiling and Selection

To navigate Tabi's optimization space, we need to properly configure the system, e.g., how many levels and which models should we use, how confident to return predictions early, and how unimportant a word is for pruning. We abstract a set of Tabi configurations as a *model candidate* with its accuracy and latency performance. A candidate is logically equivalent to other DNN serving units, e.g., single models in INFaaS [57] and ensembles in Cocktail [31], such that Tabi can be compatible with existing inference scaling systems.

Tabi includes an offline profiling and an online selection process. Profiling generates a range of differently optimized candidates. When a task arrives, Tabi selects a candidate whose *combined* performance can meet the targets with the

lowest latency. Tabi's candidate profiling and selection generally follow the SOTA inference pipeline [31, 57] while accommodating the internal mechanism of Tabi.

Tabi-aware offline profiling. The performance of a candidate is jointly decided by the base models and Tabi hyperparameters (e.g., c, α, and w). First, we profile the registered single models using representative datasets as INFaaS does. We save their statistics (e.g., accuracy and latency) and Tabi-specific intermediate variables (Figure 5), including models' softmax outputs (#class-length vectors), calibrated confidence scores (floats), and importance weights from attention (#token-length vectors). These variables add negligible storage overheads (i.e., a few megabytes per model).

Next, we obtain the combined performance of candidates with early stopping for efficiency. Tabi automatically generates a list of possible multi-level DNN combinations starting from 2 levels with accuracy in ascending order using registered models. To remove impractical choices (e.g., putting LLMs at all levels or using too many models), we only keep candidates whose 75th tail latency is within 1.5× of the slowest model used. For each combination, we test a range of hyperparameter values. For dispatcher cut-off c, testing different values does not invoke additional DNN computations because they essentially rearrange the saved softmax results. We decrease c from 0.95 (1 means always using LLMs) with step size 0.05 and stop when the combined accuracy is lower than the average of the models used. We set ensemble weight w to be proportional to each model's accuracy (see §4.3). Since word pruning modifies the input, profiling the pruning scale α requires extra rounds of DNN inference only for LLMs. We use the saved attention weights of a representative efficient model DistilBERT and increase α from 0.5 (0 means no pruning) with step size 0.1 to generate the pruning mask until an LLM's accuracy loss exceeds 2%. These designs reduce the overhead over the standard one-time profiling (discussed in §7.5). Finally, we save the metadata of Tabi candidates at the model repository just like single models.

Online candidate selection. With the profiled performance, Tabi selects candidates for online tasks using the saved metadata from the model repository following existing criteria [57]. We choose the candidate that achieves the lowest latency that can meet the performance target (accuracy and latency) of a task. Then the controller launches the selected model instances on the worker machine to serve queries of this application. Our work focuses on optimizing single data inference while existing resource and model scaling systems still apply. For example, a Tabi candidate could horizontally scale to multiple instances or vertically switch to different internal models or parameters to handle workload changes like single models (§7.5).

6 Implementation

We build Tabi using Python and libraries including PyTorch [5] as the base ML framework, HuggingFace Transformers [4] for the NLP toolbox and model implementations, and TorchServe [7] as the inference system backbone.

Clients can send text queries to the inference REST API, and developers can register trained DNNs using the management API, both hosted by the controller. We implement the model selection algorithm on the controller. Each active language model runs in a process, whether using GPU or CPU, on the worker machine. The model repository stores trained models on networked storage via GlusterFS [1] and conducts offline profiling on an available worker. It saves the metadata of candidates as Python dictionary for fast lookup and can upgrade to distributed Redis [6] for scalability.

The multi-level inference engine is a logical entity, as shown in Figure 4. The probabilistic dispatcher (§4.1), attention-based word pruning (§4.2), and weighted multi-level ensemble (§4.3) all reside in the controller, exchanging the control logic and intermediate results with the DNNs on the worker. Implementing attention-related operations uses HuggingFace standard APIs without requiring model changes. The separation of control and compute ensures the scalability of Tabi, as models for the same task do not have to run on the same physical machine. Besides latency, a monitoring process updates the CPU/GPU and memory utilization of the worker to the controller for extensible resource scheduling which is not the focus of this paper (§2.1).

7 Evaluation

We first evaluate Tabi's overall performance of reducing inference latency (including average and tail) while meeting LLM-grade target accuracy compared to SOTA inference systems (§7.2). To demonstrate the effectiveness of Tabi's design choices, we dive into the candidate profiling and selection, the three system components of the multi-level inference engine (§7.3), and the impact of hyperparameters (§7.4). Finally, we discuss the system overhead, ML-native optimizations, and generality (§7.5). The main takeaways are:

- Tabi reduces inference latency by 21%-40% on average for accuracy-demanding applications;
- Tabi minimizes the system overhead and achieves comparable tail latency performance;
- Tabi shows advantages over ML-native optimizations and is compatible with other models/systems.

7.1 Methodology

Testbed setup. We evaluate Tabi using a server with 2 NVIDIA V100 GPUs, 40 CPU cores, and 128 GB memory, running Ubuntu 18.04. The server is network-attached to hard disks as the model store. The logical controller and worker are located on the same server in our evaluation.

Model	#Parameters (million)	Latency (ms)	Accuracy (%)	*Pf*
BERT-small	28	6	72.1	6
DistilBERT* [59]	66	7	83.2	5
ALBERT* [43]	11	14	84.5	5
PruneBERT* [60]	110	15	81.2	4
DeBERTa-small	142	12	86.9	4
BERT-base [18]	110	17	84.1	4
RoBERTa-base [47]	124	19	86.3	4
DeBERTa-base [35]	184	20	88.8	3
BERT-large	340	24	86.7	2
RoBERTa-large	355	26	90.6	2
DeBERTa-large	406	29	91.3	2
DeBERTa-xlarge	886	38	91.7	1

Table 1. Summary of the language model architectures ranked by inference latency for MNLI. The latency is the DNN execution time. For other tasks/datasets, the relative performance is similar. We consider the bottom four to be LLMs because of their large sizes. A lower packing factor (*Pf*) suggests a higher per-model cost. * denotes ML-native optimized models.

Tasks, datasets, and models. We use the GLUE benchmark [72], MASSIVE [26], and CLINC150 [44]. GLUE is a standard evaluation method that covers 9 classification and regression datasets. MASSIVE (en-US) for scenario detection and CLINC150 for intent classification each have 18 and 150 classes. The validation (dev) sets we use have mostly thousands of data points. Each query contains a single data sample without zero padding (i.e., batch size = 1) and arrives after the previous one finishes. We select 12 representative and generic language models in Table 1, ranging from highly efficient to having top-grade accuracies. Three models noted with * are each optimized with knowledge distillation, parameter sharing, and pruning. Most of the fine-tuned models are publicly available [2]; if not, we fine-tune them using the pre-trained backbone.

Model storage. The size of models in Table 1 is 32bit × #parameters; the model zoo also lists models' size [2]. For example, DistilBERT (66M), BERT-base (110M), and RoBERTa-large (355M) each proportionally take 268MB, 440MB, and 1.43GB. Models fine-tuned for different tasks in Table 2 have almost the same size (<1%) since the task-specific classifier has much fewer parameters than Transformer blocks (Figure 2). Selecting from available models is a standard practice in model-less inference [57], and not all models will be deployed. Besides, inference systems are not the only user of stored models [33, 52]. We curate various models to show Tabi's generality, while the user-maintained repository size depends on users' deployment scale. Tabi is not designed for edge devices, as distributing models is much more costly [88].

Baselines. We compare Tabi with INFaaS [57] and Cocktail [31]. INFaaS selects the best single model, and Cocktail forms a voting ensemble of smaller models to reduce latency. Both systems propose resource scheduling designs, while we focus on optimizing the latency of single data inference and estimate the cost and supported throughput in a static setting. We set Tabi and Cocktail to use both GPUs to execute multiple models while INFaaS uses a single GPU.

Metrics. The main latency metric is the average DNN inference time per query, including Tabi's control logic. Data pre-processing time (e.g., tokenization) is excluded because it is a static and small factor which is highly optimized [3] but not by Tabi. We show multiple percentiles of the inference latency given the variance of tasks. The accuracy metric is the percentage of queries that obtain the correct results. We average the results of three runs. To understand the cloud serving cost on our testbed, we use the *packing factor* (*Pf*) introduced in Cocktail as a proxy of unit price, which is the number of models that can be executed concurrently on a single GPU without latency degradation (>10%). Larger models have lower packing factors, thus the per-model cost ($1/Pf$) is higher. The estimated cost of serving a dataset ($\bar{C}$) is the sum of the product of each model (m)'s cost ($1/Pf_m$), average time of the queries it serves (T_m), and percentage of workload (W_m): $\bar{C} = \sum_{m \in M} 1/Pf_m \times T_m \times W_m$. For INFaaS, there is one model, and W is 100%; Cocktail uses multiple models (M), and each W_m is 100% without auto-scaling; for Tabi, the workload handled by the small model (W_s) is 100% while the large model's $W_l < 100\%$ (see §7.3.2). We test the throughput (queries per second) using the workload pattern described before which has no queuing, batching, or scaling.

7.2 Overall Performance

Table 2 summarizes the evaluation results. We set the target accuracy of tasks to be rather high to reflect Tabi's scope of serving accuracy-demanding applications that require LLMs.

Accuracy. For all tasks, Tabi achieves the LLM-grade accuracy targets. Compared to INFaaS, which uses LLMs to serve all queries, and Cocktail, which uses an ensemble of models, Tabi's accuracy difference is within 1% on average. Tabi can balance latency and accuracy by accurately offloading the confident queries. As a result, early returned queries have a similar accuracy whether served by efficient models or LLMs (detailed analysis in §7.3.2). For the challenging queries which would have degraded quality if served by small models, Tabi utilizes LLMs and ensemble learning to ensure accuracy (details in §7.3.3 and §7.3.4).

Latency. Tabi drastically reduces the average inference latency by 21% to 40% compared to INFaaS across various tasks. Compared to the recent Cocktail, Tabi also achieves an 11%-26% average latency reduction. Figure 7 shows the detailed

	Method	SST-2	MNLI (-mm)	RTE	QQP	MRPC	CoLA	QNLI	STS-B	MASSIVE	CLINC
Tgt. acc. (%)	-	95	90	85	92	90	65	94	92	92	97
Accuracy (%)	INFaaS	96.1	91.2	86.6	92.3	90.9	67.6	94.7	92.4	92.5	97.3
	Cocktail	95.4	90.4	85.2	92.1	90.0	65.1	94.3	92.1	92.1	97.0
	Tabi	95.6	90.4	86.0	92.1	90.1	65.2	94.6	92.0	92.1	97.0
Latency (ms)	INFaaS	22.0	25.8	38.1	25.4	24.9	22.5	26.0	21.2	20.9	21.3
	Cocktail	17.8	22.9	34.7	20.8	20.2	18.2	22.3	17.5	15.4	17.2
	Tabi	13.2	20.3	30.0	16.0	16.5	15.7	18.7	13.4	13.5	14.8
Estimated cost & tput	INFaaS	11.6/42.7	13.3/36.1	19.5/24.6	13.2/36.3	13.2/36.4	11.8/40.7	13.5/35.5	11.4/41.9	11.2/42.4	11.5/41.8
	Cocktail	9.4/53.2	15.3/40.0	20.3/26.8	11.7/43.7	12.1/44.0	10.7/50.1	12.8/40.6	9.6/53.0	9.5/55.9	10.5/52.9
	Tabi	5.8/63.8	9.3/42.1	14.2/29.2	7.2/53.5	7.9/52.2	6.5/53.8	8.8/46.5	6.0/62.2	5.9/61.9	6.3/59.3
Latency reduction (%)	-	40/26	22/11	21/12	37/23	34/18	30/14	28/16	37/23	35/12	30/14

Table 2. Summary of evaluation tasks. Compared to INFaaS, Tabi achieves 21%-40% average inference latency reduction while meeting the demanding accuracy target. Compared to Cocktail, Tabi also reduces the latency by 11%-26%. In our static setting, Tabi can reduce the estimated cost by 27%-50% and 30%-39% and increase the throughput by 16%-49% and 5%-20% respectively. The costs are estimated using the packing factor (*Pf*).

latency performance. The 75% and 99% tail latencies are important to large-scale systems, while the 25% and median latencies are also crucial to ML analytics since many applications favor "freshness" when the follow-up analysis can be updated when new results arrive [21]. Tabi significantly reduces the 25% and median latencies by up to 62% and 45% compared to INFaaS and Cocktail, thus is especially effective for agile NLP applications. We also achieve better tail latency than INFaaS and close to Cocktail which expressly focuses on parallel execution. Tabi works well for tasks with 10+ classes (e.g., MASSIVE and CLINC150) since the dispatcher can well handle multi-class confidence calibration.

Estimated cost and throughput. Tabi can reduce the cloud serving cost and support higher throughput by running fewer LLMs. As explained in §7.1, we estimate the cost of models normalized by datasets using the same GPU testbed assuming the workloads and hardware are well scheduled so that the cost of running each model is relative to its usage fraction of the device. Tabi reduces the cost $\bar{C}$ by 27% to 50% and 30% to 39% compared to INFaaS and Cocktail respectively. Cocktail sometimes gets higher costs than INFaaS since more base-size models are used; Cocktail uses auto-scaling for dynamic workloads to reduce costs which is not activated with our static query pattern. For single data inference workload, Tabi generally improves the throughput; compared to INFaaS and Cocktail, we can serve 16% to 49% and 5% to 20% more queries.

7.3 Performance Breakdown

We deep dive into the performance of each system component on two representative datasets, SST-2 and MNLI, to explain our design choices. We define task difficulty as the accuracy gap between using small models and LLMs rather than the absolute accuracy. SST-2 is a relatively simple sentiment analysis task with two labels, while MNLI is a more challenging natural language inference (NLU) task with three labels where LLMs perform much better than small models.

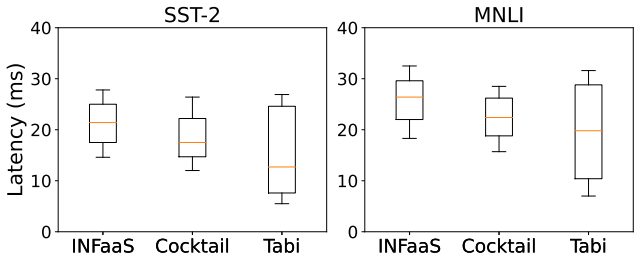


Figure 7. Latency breakdown. Each box shows the 25%, median, and 75% latencies; whiskers plot the 1% and 99% non-outlier values. Tabi largely reduces the median and 25% latency and maintains the tail latency compared to INFaaS.

7.3.1 Candidate Selection and Hyperparameters.

We find that for all the evaluated tasks, *Tabi always selects the inference engine with two levels*: an efficient DNN and an LLM, skipping the base-size models in Table 1. For example, for SST-2 Tabi selects DistilBERT and RoBERTa-large, while for MNLI, Tabi selects DistilBERT and DeBERTa-large. No candidate uses DeBERTa-xlarge for its very limited improvement over the large version, nor BERT-large over other modern architectures. Similarly, INFaaS mostly selects RoBERTa-large, and Cocktail selects multiple base and small models.

Our model profiling and selection bound Tabi's overhead; thus, it favors a two-level structure over more levels. As shown in Figure 7, using two models can strike a balance

between workload sharing (reducing average latency) and system overhead (maintaining a comparable tail latency). We analyze the performance of using three models in §7.5.

In addition to models, this step sets the Tabi hyperparameters. For example, it chooses the dispatcher cut-off threshold c of 0.95 for SST-2 and 0.85 for MNLI and pruning scale α of 0.7 and 0.9. A larger c lets the first model level return more queries with high confidence: 89% of queries of SST-2 have confidence above 0.9, while only 52% have such high confidence for MNLI. A larger α prunes more words for larger latency reduction. We test the sensitivity in detail in §7.4.

7.3.2 Workload Sharing with Dispatcher. After going through the first-level DNN, the probabilistic dispatcher reads the calibrated confidence score and decides whether to return the outputs directly. A good dispatcher should return as many easy queries as possible to produce fresh results. Meanwhile, the accuracy gap between using efficient models and LLMs should be small for early returned queries and large for re-routed ones so that they can get what they need.

For SST-2, Tabi serves 69.8% of queries only using a small DNN while achieving 97.7% and 91.4% accuracies for returned and re-routed queries. To understand this performance, we also test alternatively serving early returned queries with an LLM (as INFaaS does) and serving re-routed queries with a small model and find the accuracies to be 98.7% and 76.2%. Their 1.0% and 15.2% gaps show that we correctly assign the right DNNs to heterogeneous queries.

For the more challenging MNLI task, Tabi early returns 49.4% of the queries. This is because LLMs have a large advantage over efficient DNNs for more difficult tasks. Tabi achieves 96.1% and 85.5% accuracies for each level, while if we switch the models, the accuracy would be 96.8% and 71.3%; the gaps are ideally 0.7% and 14.2%. To summarize, Tabi achieves minimal accuracy loss (within 1%) for early returned queries, suggesting that our dispatcher can correctly distinguish the easy queries from the rest.

Figure 7 also shows the modest drawback of Tabi: Compared to the largely optimized 25% and median latencies, the tail latency does not improve much compared to INFaaS and is not as good as Cocktail. This is because Tabi sequentially runs DNNs for a limited number of queries while Cocktail always runs in parallel. That being said, our attention-aware pruning successfully reduces overheads and dismisses the concern of slowing down the tail performance of LLMs.

7.3.3 Tail Latency Reduction With Attention. By reusing the attention weights of the small DNN to prune input words, we can accelerate LLM inference of the re-routed queries and thus reduce the tail latency. Table 3 shows the performance of word pruning. For SST-2, by pruning the 14.2% least-attended words from 26 words per query on average to 22.3 words, we can reduce the execution latency of LLM by 17%. The 0.3% accuracy loss on the re-routed queries can be remedied by the weighted ensemble (see §7.3.4).

Task	Word pruning ratio (%)	LLM latency reduction (%)	LLM accuracy loss (%)
SST-2	14.2	**17**	0.3
MNLI	13.6	**15.4**	0.3

Table 3. Attention-based word pruning reduces the inference latency of LLMs with minimal accuracy loss and offsets the extra small DNN overhead in the tail performance.

For MNLI, the performance of word pruning is similar. One difference is that the NLU task of MNLI consists of two separate sentences in a query rather than a single one. We design Tabi for generality so that it can serve various input formats and tokenizers by focusing on words rather than tokens. As a result, optimizing the re-routed queries by ~14% can approximately offset the overhead of sequentially executing an efficient DNN (compared to INFaaS in Figure 7).

7.3.4 Weighted Multi-Level Ensemble. Rather than directly outputting the LLM predictions for re-routed queries, we form a weighted ensemble of multi-level models whose results are already available. We find that by weighted ensembling both levels, we can improve the LLM accuracy by 0.4% (91.4% to 91.8%) and 0.3% (85.5% to 85.8%) on these datasets without extra DNN computations, neutralizing the accuracy loss of the attention-based word pruning (see Table 3).

7.4 Sensitivity Analysis

Tabi automatically selects the hyperparameters in a candidate to achieve a balance: meeting the accuracy target while having the lowest latency. Here we analyze the accuracy-latency impact of c and α using alternative values, while the ensemble weight w is directly set by ML guidelines (§4.3).

Dispatcher cut-off c**.** Reducing c from 1 will monotonically but non-linearly increase the early-return ratio. In Figure 8, by moving c from 0.75 to 0.85 and 0.95, 92.4%, 87.4%, and 69.8% of queries in SST-2 are early returned, thus increasing both accuracy and latency. Tabi sets c to 0.95 to meet the accuracy target (95%). For MNLI, the patterns are similar: 59.4%, 49.4%, and 16.0% of queries are early returned, but the accuracy does not significantly improve when c increases to 0.95, which is why Tabi sets c to 0.85 to balance latency.

Word pruning ratio α**.** Increasing α will prune more words from LLMs' inputs, e.g., 3.8%, 14.2%, and 62.5% of words are removed using $\alpha = 0.5$, 0.7, and 0.9 for SST-2. For SST-2, there is an exception that the accuracy when $\alpha = 0.7$ is slightly higher than $\alpha = 0.5$. We assume this is because pruning a few words removes the noise in the data, and the accuracy resumes decreasing when $\alpha = 0.8$ when some useful words are affected (not shown in the figure).

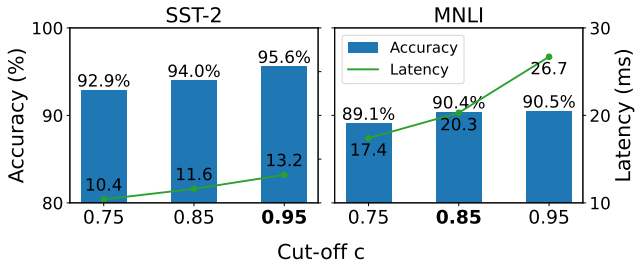


Figure 8. Impact of different c on the *combined* accuracy and latency. Selected values are in **bold**.

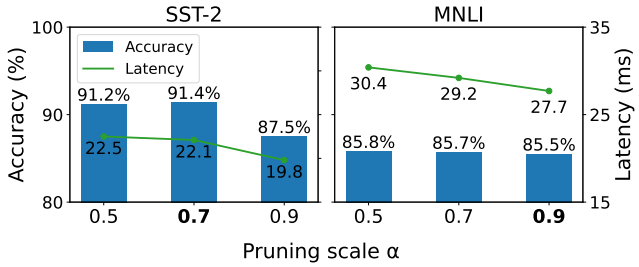


Figure 9. Impact of different α on the *re-routed* accuracy and latency. Selected values are in **bold**.

7.5 Discussion

System overheads. Tabi's major overhead is running an extra DNN for re-routed queries, which has been analyzed in detail. We find that non-DNN overheads are minimal: Tabi's control logic (e.g., Equation 4, 5, and 6) constantly takes less than or around 5ms per query, mostly on obtaining attention weights, plus small tensor (e.g., softmax logits) operations. Compared to the standard profiling, Tabi runs each model ~2.5 times on average to test α with our variable reusing design at the same level of tens of minutes as INFaaS.

Compared to ML-native optimizations. We differentiate Tabi from model compression and adaptive inference by providing LLM-grade performance in model-less inference without per-model customization. Tabi works well with optimized models (Table 1) by using them as the efficient level, exploiting their speed and making up for their accuracy losses. We evaluate an early-exit optimization, DeeBERT [80], as shown in Table 4. While DeeBERT provides two optimized base-size models out of the box which cannot meet our target accuracy, it shows 11%-40% latency reduction and <1% accuracy loss compared to its own baselines. Tabi performs better than the stock DeeBERT because early-exit requires customization expertise and fine-tuning, and thus has limited model choices. In addition, we customize and train an LLM-size DeeBERT model with RoBERTa-large, which in theory should have higher efficiency given Tabi's system design. We find that the early-exit LLM does not show a clear advantage in Table 4 because Tabi achieves extra gain through attention-based word pruning. Nevertheless, early-exit will perform better when accuracy targets are relaxed and not requiring LLMs, as discussed in the next paragraph. Users can apply a cost-benefit analyzer (e.g., LiteReconfig [83]) on top of our profiling module to choose from system-based and ML-native designs regarding the use case or incorporate it into Tabi's controller to select candidates including optimized models and benefit from Tabi's architecture.

Task	Tabi	DeeBERT-BERT-base	DeeBERT-RoBERTa-base	DeeBERT*-RoBERTa-L
SST-2	95.6/**40%**	93/**40%**	94.4/26%	**95.9**/38%
MNLI	**90.4**/22%	83.9/14%	87/19%	**90.4**/**24%**

Table 4. Accuracy (%) and latency reduction of Tabi and DeeBERT [80]. Tabi has similar performance even compared to a customized LLM. * denotes requiring ML expertise.

Accuracy (%)	Mean latency	Median latency	99% latency	Level return distribution
90.2 (-0.2%)	22.0 (+8.4%)	12.7 (-2.3%)	49.4 (+70.3%)	45.6%/36.8% /17.6%

Table 5. Compared to Tabi's two-level decision, using three models invokes the LLM less but prohibitively increases the tail latency by 70.3%, and so does the mean.

Relaxed target accuracy. For relaxed accuracy targets that do not require LLMs and are out of our scope, we can gracefully switch Tabi candidates to efficient single models. We test that the break-even points for SST-2 and MNLI are 92% and 86% when Tabi has the same accuracy/latency as a modern model, DeBERTa-small. Similarly, early-exit-optimized RoBERTa-large can achieve 3% more latency reduction than Tabi only when accuracy targets are relaxed under 91% and 87% respectively at the cost of giving up model-less inference. Therefore, early-exit works best for dedicated tasks with less strict accuracy requirements.

Using three levels. In our evaluation, Tabi rules out using three or more models for exceeding tail latency. Here we analyze the latency performance of using three models (with DeBERTa-base) in Tabi for MNLI and the same setting. We find that although the middle-sized model can further reduce the workload of the LLM from 50.6% to 17.6%, the 99% tail latency increases by 70.3% for serving challenging queries with an extra model, making token pruning irrelevant and this candidate impractical.

8 Related Work and Potential Extensions

DNN inference systems. INFaaS [57] and Cocktail [31] are all-around inference systems that have been discussed in

detail. Clipper [15] is an early generalized inference system that serves different application-level targets. PRETZEL [45], Nexus [62], and TurboTransformers [23] focus on low-level DNN execution efficiency which are complementary to our work. sensAI [73] uses pruning to decompose a large model into multiple binary classifiers to utilize more computation devices. Clockwork [30] reduces the variability of GPU inference latency by re-ordering queries based on their targets and avoiding interference. MArk [86] focuses on cost-aware resource procurement policies while managing the objectives of the task. Many research works exploit serverless cloud computing for DNN inference [8, 64]. LiteFlow [87] designs a kernel-space fast path for efficient model inference.

ML optimizations. Research on adaptive inference, including early-exit [80, 89] which reduces the depth of DNN computation, and token pruning [29, 74] which shrinks the data span, has been discussed in detail (§4 and §7.5). The similar ideas of optimizing data (i.e., words or pixels) have been discussed in CV systems [46, 76–78]. Some works focus on reducing the computational cost of the attention [66, 84] which are orthogonal to our work. New model architectures [38, 59] and compression techniques [28, 67] (e.g., quantization) try to reduce the cost of LLMs while preserving their accuracy. Tabi can serve them as candidates.

Potential extensions. Vision Transformers (ViTs) are a new group of models that adapt this NLP architecture to computer vision (CV) tasks like image classification [20]. Tabi should achieve comparable speedup when serving ViTs, since they scale up similarly to LLMs [85], and our technical dependencies, especially attention-based token pruning, have been recently explored on ViTs [39, 55]. For other models (e.g., CNNs), the benefits of Tabi will not be significant, and the tail latency may not get offset, as our designs are motivated by the scaling characteristics and the attention mechanism that are unique to Transformers.

9 Conclusion

Delivering LLMs' top-grade accuracy with model-less inference systems causes huge overheads. Tabi can reduce the latency of serving LLMs while maintaining accuracy by sharing LLMs' heterogeneous workload with efficient models based on per-query feedback. Tabi uses attention to reduce system overheads and tail latency. We evaluate Tabi on multiple datasets and find that Tabi reduces the average inference latency of LLMs by 21%-40% regarding SOTA systems.

Acknowledgment

We thank the anonymous EuroSys reviewers and our shepherd Dr. Somali Chaterji for their constructive feedback and suggestions. This work is supported in part by the Key-Area R&D Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16213621, ITF-ACCESS, the NSFC Grant 62062005, and the Turing AI Computing Cloud (TACC) [82]. Haisheng Tan is partly supported by the NSFC Grant 62132009, and Kun Guo is partly supported by the Natural Science Foundation of Fujian Province Grant No.2022J01118. We thank Yilun Jin and Han Tian for providing valuable feedback regarding the early idea. Kai Chen is the corresponding author.

References

[1] GlusterFS. https://www.gluster.org/.
[2] HuggingFace models. https://huggingface.co/models.
[3] HuggingFace Tokenizers. https://github.com/huggingface/tokenizers/.
[4] HuggingFace Transformers. https://github.com/huggingface/transformers/.
[5] PyTorch. https://pytorch.org/.
[6] Redis. https://redis.io/.
[7] TorchServe. https://github.com/pytorch/serve.
[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.
[9] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021.
[10] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017.
[11] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
[12] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.
[13] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
[14] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, 2019.
[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.
[16] Yiming Cui, Wanxiang Che, Ting Liu, Bing Qin, Shijin Wang, and Guoping Hu. Revisiting pre-trained models for Chinese natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 657–668, Online, November 2020. Association for Computational Linguistics.
[17] Shrey Desai and Greg Durrett. Calibration of pre-trained transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 295–302, 2020.
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
[19] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15, 2000.
[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani,

Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[21] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

[22] Hugging Face. Pipelines - Hugging Face, 2021. https://huggingface.co/docs/transformers/main_classes/pipelines.

[23] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[24] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[26] Jack FitzGerald, Christopher Hench, Charith Peris, Scott Mackie, Kay Rottmann, Ana Sanchez, Aaron Nash, Liam Urbach, Vishesh Kakarala, Richa Singh, et al. Massive: A 1m-example multilingual natural language understanding dataset with 51 typologically-diverse languages. *arXiv preprint arXiv:2204.08582*, 2022.

[27] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[28] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Hassan Sajjad, Preslav Nakov, Deming Chen, and Marianne Winslett. Compressing large-scale transformer-based models: A case study on bert. *Transactions of the Association for Computational Linguistics*, 9:1061–1080, 2021.

[29] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. Power-bert: Accelerating bert inference via progressive word-vector elimination. In *International Conference on Machine Learning*, pages 3690–3699. PMLR, 2020.

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

[31] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, April 2022.

[32] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

[33] Peizhen Guo, Bo Hu, and Wenjun Hu. Sommelier: Curating dnn models for the masses. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1876–1890, 2022.

[34] Abhishek Gupta. The imperative for sustainable ai systems. *The Gradient*, 2021.

[35] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

[36] Kexin Huang, Jaan Altosaar, and Rajesh Ranganath. Clinicalbert: Modeling clinical notes and predicting hospital readmission. *arXiv preprint arXiv:1904.05342*, 2019.

[37] Angela H Jiang, Daniel L-K Wong, Giulio Zhou, David G Andersen, Jeffrey Dean, Gregory R Ganger, Gauri Joshi, Michael Kaminksy, Michael Kozuch, Zachary C Lipton, et al. Accelerating deep learning by focusing on the biggest losers. *arXiv preprint arXiv:1910.00762*, 2019.

[38] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2019.

[39] Zhenglun Kong, Peiyan Dong, Xiaolong Ma, Xin Meng, Wei Niu, Mengshu Sun, Bin Ren, Minghai Qin, Hao Tang, and Yanzhi Wang. Spvit: Enabling faster vision transformers via soft token pruning. *arXiv preprint arXiv:2112.13890*, 2021.

[40] Anders Krogh, Jesper Vedelsby, et al. Neural network ensembles, cross validation, and active learning. *Advances in neural information processing systems*, 7:231–238, 1995.

[41] Volodymyr Kuleshov, Nathan Fenner, and Stefano Ermon. Accurate uncertainties for deep learning using calibrated regression. In *International Conference on Machine Learning*, pages 2796–2804, 2018.

[42] Meelis Kull, Miquel Perello Nieto, Markus Kängsepp, Telmo Silva Filho, Hao Song, and Peter Flach. Beyond temperature scaling: Obtaining well-calibrated multi-class probabilities with dirichlet calibration. *Advances in Neural Information Processing Systems*, 32:12316–12326, 2019.

[43] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[44] Stefan Larson, Anish Mahendran, Joseph J Peper, Christopher Clarke, Andrew Lee, Parker Hill, Jonathan K Kummerfeld, Kevin Leach, Michael A Laurenzano, Lingjia Tang, et al. An evaluation dataset for intent classification and out-of-scope prediction. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1311–1316, 2019.

[45] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626. USENIX Association, 2018.

[46] Xiaoxiao Li, Ziwei Liu, Ping Luo, Chen Change Loy, and Xiaoou Tang. Not all pixels are equal: Difficulty-aware semantic segmentation via deep layer cascade. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3193–3202, 2017.

[47] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[48] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150, 2011.

[49] Richard Maclin and David Opitz. An empirical evaluation of bagging and boosting. *AAAI/IAAI*, 1997:546–551, 1997.

[50] Ankur Manna, Rohit Kundu, Dmitrii Kaplun, Aleksandr Sinitca, and Ram Sarkar. A fuzzy rank-based ensemble of cnn models for classification of cervical cytology. *Scientific Reports*, 11(1):1–18, 2021.

[51] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in Neural Information Processing Systems*, 32:14014–14024, 2019.

[52] Laurel Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. Managing ml pipelines: feature stores and the coming wave of embedding ecosystems. *Proceedings of the VLDB Endowment*, 14(12):3178–3181, 2021.

[53] Titouan Parcollet and Mirco Ravanelli. The energy and carbon footprint of training end-to-end speech recognizers. 2021.

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

[55] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *Advances in neural information processing systems*, 34:13937–13949, 2021.

[56] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

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

[58] Matthew J. Salganik and Robin C. Lee. To apply machine learning responsibly, we use it in moderation - New York Times Open, 2020. https://open.nytimes.com/to-apply-machine-learning-responsibly-we-use-it-in-moderation-d001f49e0644.

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

[60] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning. 2020.

[61] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

[62] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[63] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 761–769, 2016.

[64] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[65] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[66] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019.

[67] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4323–4332, 2019.

[68] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[70] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, 2019.

[71] Mariko Wakabayashi. Speeding up transformer CPU inference in Google Cloud - Twitter, 2021. https://blog.twitter.com/engineering/en_us/topics/insights/2021/speeding-up-transformer-cpu-inference-in-google-cloud.

[72] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, 2018.

[73] Guanhua Wang, Zhuang Liu, Brandon Hsieh, Siyuan Zhuang, Joseph Gonzalez, Trevor Darrell, and Ion Stoica. sensai: Convnets decomposition via class parallelism for fast inference on live data. *Proceedings of Machine Learning and Systems*, 3, 2021.

[74] Hanrui Wang, Zhekai Zhang, and Song Han. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[75] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems*, 33:5776–5788, 2020.

[76] Yiding Wang, Weiyan Wang, Duowen Liu, Xin Jin, Junchen Jiang, and Kai Chen. Enabling edge-cloud video analytics for robotics applications. In *Proceedings of the IEEE International Conference on Computer Communications, Virtual Conference*, pages 10–13, 2021.

[77] Yiding Wang, Weiyan Wang, Duowen Liu, Xin Jin, Junchen Jiang, and Kai Chen. Enabling edge-cloud video analytics for robotics applications. *IEEE Transactions on Cloud Computing*, 2022.

[78] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *HotCloud*, 2019.

[79] Lilian Weng. Attention? attention! 2018. https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html.

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

[81] Hu Xu, Bing Liu, Lei Shu, and S Yu Philip. Bert post-training for review reading comprehension and aspect-based sentiment analysis. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2324–2335, 2019.

[82] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[83] Ran Xu, Jayoung Lee, Pengcheng Wang, Saurabh Bagchi, Yin Li, and Somali Chaterji. Litereconfig: cost and content aware reconfiguration of video object detection systems for mobile gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 334–351, 2022.

[84] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

[85] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12104–12113, 2022.

[86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1049–1062, 2019.

[87] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 414–427, 2022.

[88] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–93, 2021.

[89] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33, 2020.