PAME: Precision-Aware Multi-Exit DNN Serving for Reducing Latencies of Batched Inferences

Shulai Zhang
Shanghai Jiao Tong University
zslzsl1998@sjtu.edu.cn

Weihao Cui
Shanghai Jiao Tong University
weihao@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
chen-quan@cs.sjtu.edu.cn

Zhengnian Zhang
Shanghai Jiao Tong University
65358998@sjtu.edu.cn

Yue Guan
Shanghai Jiao Tong University
bonboru@sjtu.edu.cn

Jingwen Leng
Shanghai Jiao Tong University
leng-jw@sjtu.edu.cn

Chao Li
Shanghai Jiao Tong University
lichao@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

ABSTRACT

In emerging DNN serving systems, queries are usually batched to fully leverage hardware resources, and all the queries in a batch run through the complete model and return at the same time. According to our findings, some queries only need to pass through a portion of the DNN model to attain sufficient precision in a DNN service. These queries can have shorter latencies if they can return early in the middle of a model. Therefore, we propose precision-aware multi-exit inference serving, PAME, to achieve the above purpose. PAME provides a holistic scheme to build a multi-exit DNN model and a corresponding system-level design of the inference engine. We use representative CV and NLP benchmarks to evaluate PAME. PAME is adaptive to various DNN tasks and service loads. Experimental results show that PAME reduces 39.9% average latency without increasing the tail latency, while maintaining 99.68% precision of the original single-exit DNN models on average.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; • **Computer systems organization → Real-time systems**.

KEYWORDS

Deep neural networks, batch inference, multiple exits

ACM Reference Format:
Shulai Zhang, Weihao Cui, Quan Chen, Zhengnian Zhang, Yue Guan, Jingwen Leng, Chao Li, and Minyi Guo. 2022. PAME: Precision-Aware Multi-Exit DNN Serving for Reducing Latencies of Batched Inferences. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524059.3532366

1 INTRODUCTION

Modern intelligent user-facing services, such as online image recognition [1, 42], online translation [2, 3], often adopt Deep Neural Networks (DNNs) as the backend to satisfy the high requirements on the accurate inference. When a large number of users may submit inference requests concurrently, the inferences are often processed in batches on high-performance accelerators (e.g., GPUs) [6, 16, 18, 42]. The inferences are batched because a single inference query cannot fully utilize the hardware. The batching mechanism significantly increases the supported load of the accelerator.

Current accelerators and DNN serving systems treat all the inference queries equally, and all the queries in a batch return simultaneously. In general, a complete and complex DNN model is often trained for a user-facing service to handle the most complex queries. However, different efforts are required for the inference queries to achieve sufficient precision, as these inference queries often have different inputs [26, 28, 52]. Complex DNN models are often wasteful for simple and canonical queries. Some queries can achieve sufficient precision by running only a part of the complex DNN model and exiting early [30, 32]. There is the opportunity to reduce the response latency of inference queries by allowing some of the queries in a batch to exit earlier.

It is nontrivial to take the above opportunity due to the complex structures of DNN models. In general, a DNN model has two main components: the *backbone*, and the functional *head*. The backbone (e.g., ResNets [27] for computer vision tasks and BERT [21] for natural language processing tasks) provides representative features, and the head generates the final results based on the features from the backbone. An inference query is not able to exit in the middle of the backbone without going through the head. It is also not applicable to directly "jump" from the middle of the backbone to the head, as the structures of the intermediate features (e.g., the dimensions of the features) from the middle of the backbone may not be compatible with the requirements of the head.

Based on the above analysis, Figure 1 shows the design principle of a DNN serving system that allows some queries in a batch to exit earlier. In general, the inference queries form a batch, enter the backbone together, and may exit from different exits. At each exit, a *bridge* structure is added to transform the intermediate features into the required form of the head. Besides, a *head* structure and

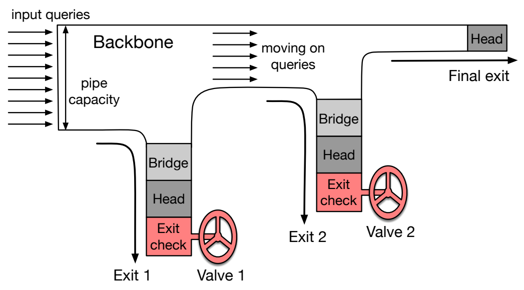

Figure 1: The principle of a multi-exit DNN serving system.

a *exit check* structure are required at each exit. The *head* structure calculates the output of each query, and the *exit check* structure determines whether an query can exit. Generally, the entire serving system acts like a multi-exit pipe; the *exit check* structures act like valves and determine the exit ratio of the samples.

In practice, designing a multi-exit batch inference system following the above principle faces three key challenges: 1) How to design the exits, as the features at different exits do not have identical structures, thus effort has to be paid to transform them into acceptable structures. 2) Where to attach the exits, as many exits can be attached on the DNN models but not all can provide satisfying performance. 3) The added exits increase the latency of the inference queries that do not exit early. Too many exits bring heavy latency overhead.

Exit Structure: With each exit demonstrating a fully functional network route, all exits should contain task-specific heads in order to obtain final results. Besides, the structures of the possible bridge, the head, and the exit check at each early exit need dedicated design. In order to attach applicable exits on the original pipe, we also have to promise the precisions of early exits to be comparable with the original single exit. Thus, the parameters of exits have to be redistributed through special training schemes and the opening of valves also requires meticulous design.

Exit Placement: The architectural design space for a multi-exit network has numerous configuration parameters, among which the exit position and the number of exits determine where to place exits. It is computationally challenging to find out the optimal network architecture because the complexity of the search space is huge.

Exit Overhead: When inferences are executed in a batch, the additional computation of exits may bring overhead and extend the latency of inferences that do not exit early. To decide where a sample should exit, fetching and analyzing intermediate results is inevitable. Such additional computation separates the complete inference process into several stages, breaks the possible original graph optimizations [9, 14, 19], requires extra kernel invocations and data movement.

The added exits should not increase the latency of the inference queries that exit normally, in other words, the tail latency of queries within a batch. In this case, the exit overhead directly affects the selection of exit structure and exit placement. There are some prior algorithm works on adding multiple exits on a DNN model for efficient inference [11, 29, 30, 49, 54]. The precision-latency trade-off is also discussed in [33, 34, 43]. However, there is a lack of consideration of the multi-exit design in batch inference scenarios. With current batching mechanisms, even if a sample can achieve the required precision early as designed in prior work, it must return simultaneously with other inferences in a batch.

To this end, we propose **PAME**, a scheme to build up a multi-exit batch inference system. The optimized multi-exit network seeks to minimize the inference latency without degrading the inference precision. To add an exit, PAME imitates the structure of the backbone to construct the bridges, applies a customized training scheme to redistribute the parameters in heads and applies a rule-based exit check method which can be applied widely to various DNN tasks (Section 5). To identify the appropriate place to attach the exits, PAME organizes a rehearsal with collected real query samples for profiling the performances of different configurations (Section 6).

The main contributions of this paper are as follows:

- A holistic mechanism to attach multiple exits on a DNN model and corresponding exit policies. The design promises the multi-exit model's precision to rival the original DNN model's precision.
- A runtime precision-aware multi-exit DNN serving system that executes batched inferences while allowing inferences to exit asynchronously. The design takes the opportunity of reducing latency for the queries with easy inputs.
- A method of determining the to-be-used exits and adapting to load changes based on input patterns. With the method, the tail latency of a batch is not increased, even if the newly added exits bring overhead.

We implement PAME with TensorRT [9]. Our experimental results on an Nvidia V100 GPU show that PAME reduces 39.9% latency of the benchmarks on average while preserving 99.68% precision.

2 RELATED WORK

In this section, we discuss the algorithm work on developing efficient multi-exit DNNs, and the system work on improving the batching policy during DNN inference.

2.1 Multi-exit DNN Models

Many algorithm works have proposed multi-exit DNN architectures for efficient inference. The principles followed for attaching exits and the principles for selecting samples to exit are two bases for multi-exit model inference in existing work.

The principles of attaching exits vary. The mechanisms in [25, 37, 49, 54] directly place exits after each block of the transformer model, assuming the overhead of exits is small. In [12, 22, 29, 30, 43], the placement of exits is hand-crafted and depends on the model architecture. Considering the overhead exits may bring, the mechanisms in [23, 33] use computational budget (e.g., the number of FLOPs) and HAPI [34] uses latency budget to decide where to attach exits to balance the accuracy/computation trade-off. Although these works achieve significant computation or latency reduction, they are not feasible when queries are organized in batches. To the best of our knowledge, PAME is the first work that realizes and illustrates the superiority of multi-exit in practical batch inference scenarios.

The principles for selecting samples to exit can be categorized into learnable exit policies and rule-based exit policies. Learnable exit policies require hand-crafted and trainable networks to decide whether a query can exit [31, 46, 47], while rule-based exit policies only check the prediction confidence at each exit to make the decision. With most existing works focusing on classification tasks and sporadically on other tasks (e.g., semantic segmentation [33]), the potential of rule-based exit policies has not been fully exploited. PAME provides a holistic methodology and implementation of rule-based exit policies for various tasks.

2.2 Batching Policies for Queries

Many works have suggested using the batching approach to improve the DNN serving system. Clipper [16] and Tensorflow Serving [42] use a batching method to group inferences received at the same time into a single batch and process them all at once. Triton [6] and Ebird [17] improve the batch-based approach by enabling concurrent execution of multiple DNNs. While these studies do not concern the intrinsic properties of DNNs, they do provide us with the potential to lower latency.

There are also other researches looking into how to improve batch inference using the information of network architecture. Padding is required to batch the inferences because RNN-based models accept input with varying sequence lengths. Cavs [50], Dynet [40], TensorFlow Fold [39], and BatchMaker [24] all aim to reduce the latency for RNN-based models owing to padding. These researches are orthogonal to PAME since they do not perceive the precision information to minimize the latency.

3 MOTIVATION

In this section, we illustrate the opportunity of reducing latency of DNN inferences through early exiting, and analyze the challenges in taking the opportunity.

3.1 Opportunity of Early Exits

In this subsection, we perform experiments to show the inference precision of various benchmarks when the queries exit early, from the middle of DNN backbones. Except for the common classification tasks, we explore more complex tasks such as pose estimation and semantic segmentation. The precision metric is the mIoU (Mean Intersection over Union) for the semantic segmentation task and the prediction accuracy for other tasks. The details of the tasks, their corresponding backbones, heads and loads are listed in Section 7.1.

Popular DNN backbones (e.g., ResNet [27] and BERT [21]) often comprise identical structures (referred to be *blocks* in this paper) to compress and optimize features. Residual blocks in ResNet series and the hidden layers in the encoder of transformer series are such blocks. There are $\{3, 4, 23, 4\}$ repetitive blocks for four regions separately in ResNet101 [27] and 12 repetitive blocks in BERT-base [21]. In the experiments, we attach exits at the end of each *block* in the backbones, we train the exits as we will discuss in Section 5.2 and we measure the inference precision at different exits as shown in Figure 2.

In the figure, the horizontal lines show the inference precision of the original DNN models, and the x-axis shows the exit position. As observed, the precision of some early exits is close to the original precision. For instance, if the exit after the 6-th block is adopted for *SST-2*, the precision can reach 98% of the final exit's precision but the computation is halved. We can also observe that, for *Cityscapes*, the precision is even 3% higher than the original precision if the

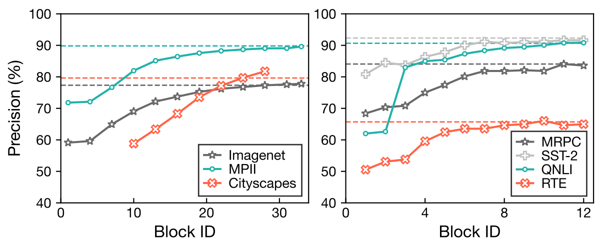

Figure 2: Precisions of the early exits on backbones (Left: ResNet101, Right: BERT-base)

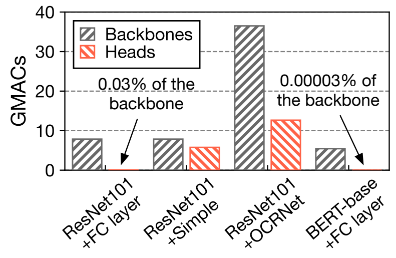

Figure 3: Comparison of MACs (Multiply-Accumulate Operations) between backbones and heads.

queries exit earlier after the 28-th block. In this case, using only part of the whole model is more efficient and can also achieve satisfying results.

Considering the precision, it is feasible to allow some inference queries in a batch to exit earlier.

3.2 Challenges of Adding Exits

It is nontrivial to add multiple exits into a DNN model for batch inference for several reasons. First, as shown in Figure 1, it is challenging to design the *exit check* structure for each exit, as the structure needs to quickly determine whether each inference query within a batch can return. The *exit check* structures vary for different tasks. In addition, adding too many exits (e.g., adding an exit at the end of each *block*) may result in the long latency of the queries that do not exit early, since a *bridge*, a *head*, and an *exit check* structure are all required in each exit.

Figure 3 shows the numbers of MACs (Multiply-Accumulate Operations) in the backbone and the head of various DNN tasks. As shown in the figure, the sizes of heads are various for different tasks and can be relatively large. For example, when the input image shape is (3, 224, 224), the OCRNet [45] (head for semantic segmentation) requires 12.64 GMACs (12.64×10^9 MACs) which is 34.6% of the backbone and the Simple [48] (head for pose estimation) requires 5.77 GMACs which is 73.5% of the backbone. The computational overhead of the heads in exits can not be overseen.

Worse, the added exits invalidate the graph optimizations which can only be applied when the model is compiled as a whole, and interrupt the inference process. Restarting the remained model requires additional kernel invocation and data movement. For instance, it is possible to add 11 exits for a DNN model with the BERT backbone that has 12 blocks. While adding 11 exits only introduces 0.0003% additional multi-add operations in total, the inference time

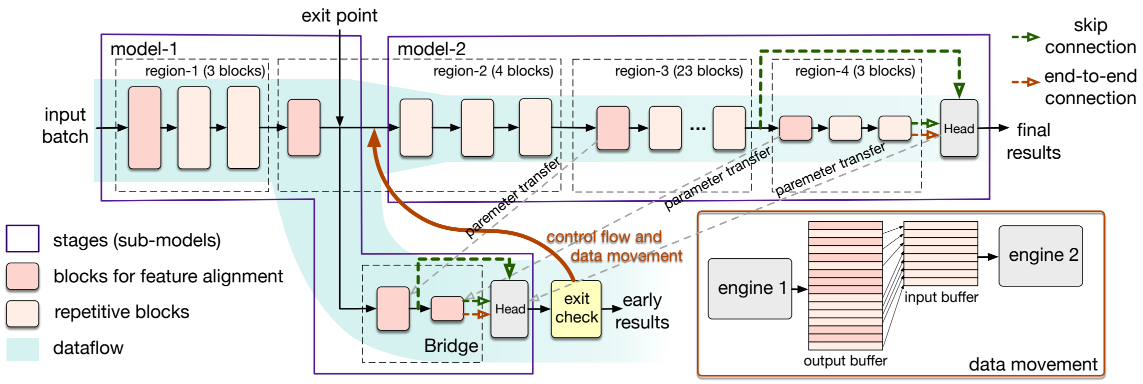

Figure 4: The design of PAME, illustrated by an example of a two-stage batch inference with ResNet101 as the backbone.

of the queries that arrive at the last exit increases from 16.75 ms to 20.03 ms, with the batch size 16.

In multi-exit batch inference, it is also possible that the latency of inference queries that do not exit early does not increase. This is because if fewer queries are processed in the later part of the DNN model, the time needed to run the later part of the model reduces. There is a trade-off between the reduced computation from the fewer queries and the increased computation brought by the exits. To this end, the placement of exits should be determined by the input pattern, as the pattern impacts the number of inference queries that can return at each exit. It is also worth noticing that, since the batch size may be reduced after each exit, the inference engine has to handle the dynamic batch size efficiently.

PAME has to consider the input pattern of the inference queries, as well as the computation of the exit structure when building the multi-exit DNN model for reducing the inference latency.

4 DESIGN OF PAME

We design PAME to build a multi-exit inference network for reducing the latency of queries. The multi-exit DNNs are trained following two constraints. First, the modification to the network shall not harm the task precision. Second, for a batch of inference queries, its tail latency in a multi-exit model is not increased compared with its tail latency in the original case.

The inference engine is also redesigned to handle the dynamic batch sizes. Specifically, we slice models into multiple stages and each stage contains an exit. Then the leaving of samples would not cause interference between different stages within a batch and would not change the batch size in each stage.

4.1 Building a Multi-exit Model

PAME does not build a new multi-exit DNN model from scratch, but transforms an existing single-exit DNN model into a multi-exit DNN model. The transformation is done as follows.

(1) PAME identifies the exits equidistantly along the backbone as exit candidates. For each exit candidate, the bridge structure is designed based on the structure of the intermediate features at the exit. PAME trains all the parameters of the multi-exit network (the backbone, bridges, and heads) together (Section 5).
(2) PAME selects the actual exits from the exit candidates based on the input pattern and the overheads of each exit (Section 6). The input pattern impacts the number of queries that may exit from each exit with enough precision. In each step, PAME finds the very exit that brings the shortest average latency for the queries without increasing the latency of the queries that go to the final exit. The found exit breaks the DNN model into two stages. PAME recursively applies the same algorithm to break the later stages.

In our design, the exit candidates can only be placed at the end of the *blocks*. We use blocks as the granularity to add exits, because the block is the basic unit/step to compress information to a low dimensional space mathematically [13]. In addition, many graph optimizations are done between layers in the same block [14, 53]. Adding an exit in a block invalidates the graph optimization, and incurs the high cost of data movement between stages, as there are high data dependencies within a block.

4.2 Serving with a Multi-exit Model

Figure 4 shows an example of serving an inference batch on a two-exit DNN model with *ResNet101* as the backbone. In the figure, the original DNN model is divided into two stages by the newly added exit at the exit point. We use $M_{0\to N}$ and H_N to represent the backbone and the head of the original single-exit model separately, and N is the number of blocks in the DNN model.

Let α represent the ID of the block of the exit in Figure 4. In this case, the original model is divided into model-1 ($[M_{0\to\alpha}+H_\alpha]$) and model-2 ($[M_{\alpha\to N}+H_N]$). H_α is the newly attached exit. PAME runs model-1 and model-2 using different inference engines, and the two parts communicate through the control flow and data movement.

PAME serves a batch of queries as follows. 1) The entire batch with batch size n is enqueued to the inference engine for model-1. 2) When model-1 completes, the exit check structure after H_α reads the output data of all the queries, and determines the queries that can exit. The queries that have high enough precision return, and the other queries proceed to model-2. The number of queries which moved on to model-2 is b. 3) The b queries' intermediate features are copied from engine 1's output buffer to engine 2's input buffer. 4) Engine 2 runs model-2 with batch size b and then returns the final results.

Since later stages may have small batch sizes, the GPU resource may not be fully utilized. In this case, stages with small batch sizes can be concurrently executed with the early stages of other batches

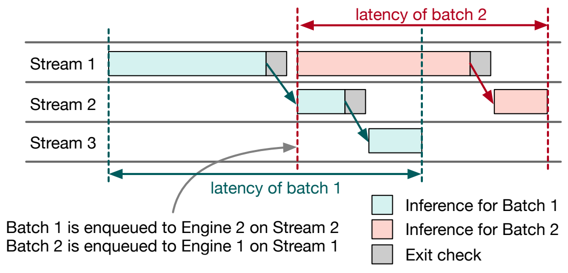

Figure 5: Parallel batch scheduling to reduce queuing time.

as depicted in Figure 5. However, the co-location may slow down the current batch. Therefore, we use an empirical batch size threshold τ_b to estimate if loading new batch would benefit. If the number of moving-on queries is lower than τ_b, a new batch can be enqueued to Engine 1 on another stream. With the parallel batch scheduling, later batches may start earlier and have a shorter queuing time.

5 THE STRUCTURE OF AN EARLY EXIT

In a newly added exit, a bridge transforms the intermediate features to the form required by the head, a head performs the inference based on the output of the bridge, and an exit check structure determines whether each query can exit with potentially high precision.

5.1 The Bridge in an Early Exit

Aiming at introducing fewer new model structures, it is intuitive to imitate the original backbone to construct bridges. For an exit, its bridge is initially identical to the entire backbone after the exit point to promise connectivity to the head. Bridges are not required for BERT-based DNN models, because the backbone can be directly connected to the heads to form exits.

It is crucial to minimize the bridge overhead, while still making sure the bridge can achieve its basic functionalities. In general, PAME should drop as many blocks in a bridge as possible while promising the bridge can still transform the feature maps to the required form of the head.

A DNN backbone is often composed of several *regions* [21, 27, 38] and blocks within a region have identical structures of intermediate features. Based on this finding, in the bridge of an exit, we only keep the first block in each region after the exit point. In this way, the bridge is capable of transforming the intermediate feature map to the required form of the head.

Take the *ResNet101* backbone in Figure 4 as an example. In ResNet101, the feature map at the exit point has a dimension of $[bs, 512, h/8, w/8]$. It is not compatible with the input dimension of the head, which is $[bs, 2048, h/32, w/32]$. Thus, the bridge is initially identical to the backbone after the exit point, and then drops all the repetitive blocks except the first block in region-3 and the first block in region-4 for the feature alignment.

Note that the bridge also supports the skip connections, as skip connections are required [36, 45] in some tasks (especially CV tasks with CNN backbones). Skip connection means that the head receives features from different regions in the backbone. For example, the OCRNet receives features from both region-3 and region-4 (green dash arrows in Figure 4). Since the bridge consists of at least one block in different regions, it can also provide the features that the head requires, thereby achieving skip connections.

5.2 Training the Heads in Exit Candidates

After determining the bridge structure, the parameters within an exit should be determined. Since exits are selected from exit candidates according to their performances, we first determine the exit candidates and then train them to achieve their best precision.

It is redundant to regard all possible exits as candidates because the performances between adjacent exits are similar. Moreover, with many exits trained together, the parameters among exits affect each other and make the training process unstable [35]. Thus, we only select K exits equidistantly along the backbone as candidates. For the i-th exit candidate, the corresponding block ID is $e_i = ai + b, i \in [1, K]$, so there are a blocks between two adjacent exits and the first exit candidate's block ID is $a + b$. Then in each training epoch, all parameters in $[M_{0\rightarrow N} + H_{e_1} + H_{e_2} + \cdots + H_{e_K} + H_N]$ are trainable.

To shorten the time for convergence, we initialize the parameters of exits by parameter transfer as shown in Figure 4. It is intuitive and convenient to transfer the parameters in the original backbone to the bridges and heads in the exit for initialization. Since the parameters have already been trained in pre-trained backbones (ResNets trained on *Imagenet* [8] or BERT trained on vast English text data [4]), we only need to fine-tune the parameters to obtain satisfying models.

The fine-tuning is done offline with the corresponding training dataset in each task. The training loss is computed as the weighted average loss of all heads. Since we have no priori preference for those exits, the weights of loss are identical.

5.3 Defining the Exit Check Structure

For an exit, its exit check structure decides whether an input query should return from the exit or not. The decision is made based on the output values of the head, as the output values already include much high-level semantic information for the corresponding DNN model. For instance, the outputs of the head in a classification model show the probability values of different labels. Based on such semantic information, PAME is able to identify whether the result of an input query is confident enough to return.

In general, a head often generates high-dimensional outputs. For instance, in the pose estimation task [10], the outputs of the head are 16 heatmaps of size $(96, 96)$, representing the probabilities of 16 human knots. The exit check structure should be able to determine whether a query can return based on the high dimensional outputs. Figure 6 shows the rule-based policy used to achieve the above purpose. The exit check structure incorporates a dimension reduction step and a threshold check step.

Both the dimension reduction and the threshold check are DNN task-dependent, as the outputs of the heads are different for the tasks. In this case, the exit check is customized for DNN tasks based on their semantics. In principle, the dimension reduction finds the principal values among all output values. It reduces the size of outputs, and eliminates irrelevant dimensions for the threshold check; The threshold check judges whether the principal values can show sufficient confidence for the query to exit. If the quality and quantity of the selected principal values are high, there is

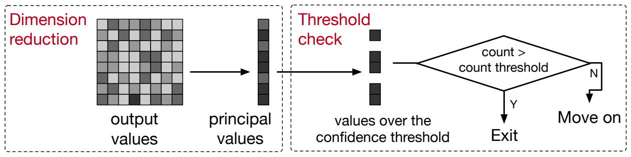

Figure 6: The rule-based policy in the exit check structure.

sufficient confidence that the query can achieve sufficient precision already [33, 54]. Thus, a confidence threshold and a count threshold are used in the threshold check. For each query, we first count the principal values that exceed the confidence threshold. If the number exceeds the count threshold, we allow the corresponding query to exit as shown in Figure 6.

As examples, we describe the exit check structure for classification, pose estimation, and semantic segmentation tasks. For classification tasks, the outputs are probabilities of labels. During the dimension reduction, the maximum probability value is regarded as the principal value. During the threshold check, the principal value should exceed the confidence threshold to let the query exit. Since there is only one principal value in classification tasks, there is no need to apply the count threshold. For the pose estimation task that aims to obtain a regression of human pose [10], the outputs of the head are 16 heatmaps of size (96, 96). During the dimension reduction, the principal values are the 5 highest probabilities from all pixels in each heatmap. For the semantic segmentation task that aims to classify every pixel of the image on *Cityscapes* [15], the outputs of the head contain 19 label probabilities for every pixel in the image. During the dimension reduction, the principal values are the highest probabilities for every pixel. During the threshold check for both the pose estimation and the semantic segmentation tasks, the quality and quantity of the principal values are compared with the confidence threshold and the count threshold, separately.

The exit check structures for different exits in a DNN task are the same, except for the confidence threshold and the count threshold. The thresholds change with the input load patterns.

6 DETERMINING THE TO-BE-USED EXITS

As the load pattern of the inputs changes, the numbers of queries that have enough precision at different exit candidates vary. To this end, PAME uses the actual load of a multi-exit model to determine the to-be-used exits from the exit candidates.

6.1 Steps of Determining the Exits

Since there are many exit candidates and each candidate has two thresholds (Section 5.3) that can be configured, there is a large number of applicable combinations of the exits. Suppose there are K exit candidates and each threshold has 10 possible values. The number of possible exit combinations is $(1 + 10 \times 10)^K$. The search space is huge if we consider all the combinations.

To resolve the large search space problem, PAME determines the exits one by one. Specifically, to determine the first exit in a DNN model, PAME first identifies the threshold configurations for each exit candidate, as if it is the only exit (Section 6.2). The threshold configurations ensure that the overall precision is satisfactory no matter which exit is attached. Next, PAME profiles the latencies of the exit candidates (Section 6.3). The exit candidate that results in the shortest average latency without increasing the tail latency is selected to be the first exit (Section 6.4). After the first exit is determined, PAME performs the above three steps iteratively on the part of the DNN model after the exit to determine other applicable exits.

By adopting the above method, the searching complexity is reduced to $O(10 \times 10 \times K)^2$. The iterative method can still find the near-optimal exit combinations.

6.2 Precision-aware Candidate Configuration

We capture the real historical load of the DNN service to be a dataset D for determining the thresholds in this step. The thresholds of an exit candidate should not be configured beforehand based on the training dataset, as they directly control the moving-on ratio (the ratio of queries that move on to later stages) after each exit, thereby influencing the overall precision of real load. Based on the real load, PAME finds the thresholds for an exit candidate that can achieve the required overall precision and minimize the moving-on ratio to minimize the average latency.

In this step, we profile the overall precision and the moving-on ratio for each combination of thresholds in each exit candidate. Let H_α represent an exit candidate and $\mathbf{p}^\alpha$ represent the exit candidate's confidence threshold and the count threshold. To calculate the corresponding overall precision with the configuration on the real dataset D (denoted as $Q^\alpha(\mathbf{p}^\alpha)$), we collect the ratio of queries that return from the exit, $r_e^\alpha(\mathbf{p}^\alpha)$. Equation 1 calculates the overall precision in this case. In the equation, $Q_e^\alpha(\mathbf{p}^\alpha)$ is the average precision of queries return from H_α, and $Q_m^\alpha(\mathbf{p}^\alpha)$ is the average precision of the moving on queries.

$$Q^\alpha(\mathbf{p}^\alpha) = r_e^\alpha(\mathbf{p}^\alpha) \times Q_e^\alpha(\mathbf{p}^\alpha) + (1 - r_e^\alpha(\mathbf{p}^\alpha)) \times Q_m^\alpha(\mathbf{p}^\alpha) \quad (1)$$

For the exit candidate H_α, we select the $\mathbf{p}^\alpha$ for which $Q^\alpha(\mathbf{p}^\alpha) > \eta Q_{\mathbf{ori}}$ and $r_e^\alpha(\mathbf{p}^\alpha)$ is maximized. $Q_{\mathbf{ori}}$ is the prediction precision of the original single-exit model. η is the task precision tolerance and $0 < \eta \leq 1$. Figure 7(a) shows an example of determining the thresholds for the pose estimation DNN task. In the figure, the z-axis is the overall precision.

The overall precision will be changed when new exits are attached. Then, if exit H_β is attached after H_α with thresholds $\mathbf{p}^\beta$, $Q_m^\alpha(\mathbf{p}^\alpha)$ will be updated by $Q_m^\alpha(\mathbf{p}^\alpha) = r_e^\beta(\mathbf{p}^\beta) \times Q_e^\beta(\mathbf{p}^\beta) + (1 - r_e^\beta(\mathbf{p}^\beta)) \times Q_m^\beta(\mathbf{p}^\beta)$ and $Q^\alpha(\mathbf{p}^\alpha)$ will also be updated accordingly. We compare the updated $Q^\alpha(\mathbf{p}^\alpha)$ with $\eta Q_{\mathbf{ori}}$ to decide whether to keep H_β.

6.3 Characterizing the Inference and the Load

The above step finds the exit candidates and their thresholds to achieve the required precision. However, the added exits may result in the long latency of the queries that do not exit early.

For each exit candidate, it is intuitive to profile and record latencies directly with the simulated load. However, adopting the naive profiling method, the latency overhead has to be profiled again to re-determine the optimal exit placement, whenever the hardware's utilization, the query load or the thresholds change.

If the batch size for each inference stage is fixed, the inference time of a batch will not be affected by the inputs of the queries. In

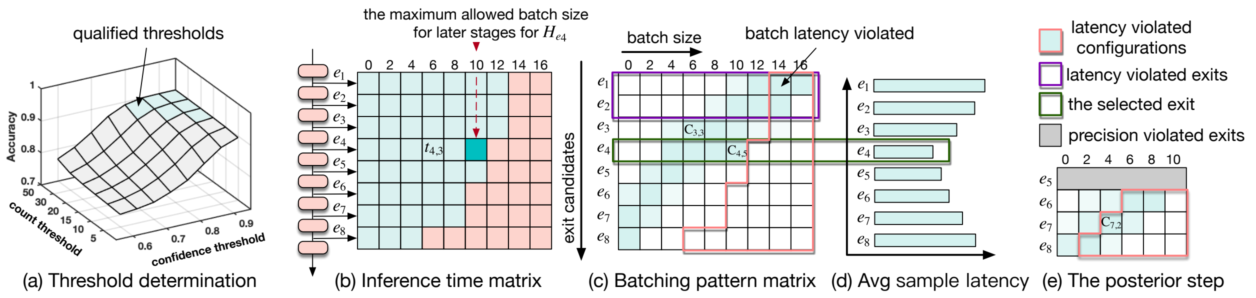

Figure 7: Illustration of the recursive exit searching procedure.

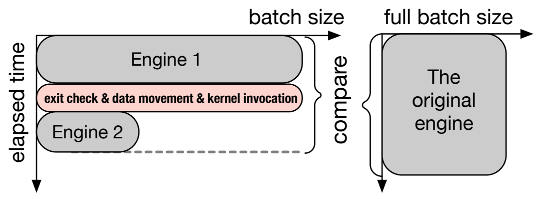

Figure 8: The elapsed time of the two-stage inference, and the single-stage inference.

this case, we decouple the profiling of the inference time at different batch sizes and the input loads to get rid of the cumbersome profiling. We use an *inference time matrix* to record the two-stage inference latency, and use a *batching pattern matrix* to record the batching pattern of loads for each exit candidate. When the hardware changes, we only need to update the inference time matrix; and when the input pattern changes, we only need to update the batching pattern matrix. The profiling effort is greatly reduced.

6.3.1 Inference Time Matrix. To identify feasible batch size configurations in each stage, we profile the query latency with the inference time matrix. Given a model $[M_{0\to N} + H_N]$ and an assigned batch size n, we first record its $n-$batched inference time t_0^n as the original tail latency.

Figure 7(b) shows an example of the inference time matrix. For the exit candidate at exit point e_i ($i \in [1, K]$), we profile the latency of the two-stage inference, including the n-batched inference of $[M_{0\to e_i} + H_{e_i}]$ and the n_j^*-batched inference of $[M_{e_i\to N} + H_N]$. The batch size of the second stage $n_j^* = jv$, where v is the profiling batch size interval, and n_j^* is not larger than the batch size of the first stage. For instance, in Figure 7(b), the $t_{4,3}$ represents the elapsed time in executing $[M_{0\to e_4} + H_{e_4}]$ with batch size $n = 16$ and $[M_{e_4\to N} + H_N]$ with batch size $n_3^* = 3v = 6$. The elapsed time of the two-stage inference also includes the time consumed in exit checking, data movement and kernel invocations for the second engine as shown in Figure 8.

In each iterative searching step, the inference time matrix records the latency of the two-stage inference beginning from the previously selected exit. It is important to decide the batch size of the first stage properly. In the first searching step, the batch size of the first stage is n. In later steps, the batch size of the first stage for profiling is the maximum allowed number of samples remaining from the previous exit. As the example in Figure 7(b), if exit H_{e_4} was picked in the first step, the batch size of the first stage in the posterior step would be $n_4 = 10$. Then the inference time matrix should update $t_{i,j}$ as the time that executing $[M_{e_4\to e_i} + H_{e_i}]$ with batch size $n_4 = 10$ and executing $[M_{e_i\to N} + H_N]$ with batch size n_j^*, where $i \in (4, K]$ and $n_j^* \le n_4 = 10$.

6.3.2 Batching Pattern Matrix. The batching pattern matrix is generated from the real historical load D. We separate D into k batches, collect the number of moving on samples in each batch, and record it in the batching pattern matrix. Figure 7(c) shows an example of the batching pattern matrix. In the matrix, each row is actually a histogram to record the number of batches. The size of the batching pattern matrix is the same to the size of the inference time matrix.

In Figure 7(c), the batching pattern matrix reveals that among k batches, $C_{3,3}$ batches allow $[6, 8)$ samples to exit from H_{e_3} and $C_{4,5}$ batches allow $[10, 12)$ samples to exit from H_{e_3}. In the posterior step of the recursive searching, Figure 7(e) shows that there are $C_{7,2}$ out of k batches allow $[4, 6)$ samples to exit from H_{e_7} and those are the samples that did not exit from the last selected exit H_{e_4}.

6.4 Latency-Aware Exit Determination

After obtaining the profiling results, we select the first exit with the shortest average query latency as shown in Figure 7(d). With exit H_{e_i} attached, the average two-stage inference latency is denoted as $T_i^{\textbf{avg}}$, and its expectation is estimated with the inference time matrix and the batching pattern matrix as shown in Equation 2.

$$E(T_i^{\textbf{avg}}) = \frac{1}{nk} \sum_{j=0\to h} ((n - n_j^*)t_{i,0} + n_j^* \cdot t_{i,j}) \cdot C_{i,j} \tag{2}$$

Before determining the exit, we first eliminate the exit candidates with which the tail latency is violated. The original tail latency t_0^n is reflected on the boundary in the inference time matrix. To obtain the boundary, each element in the inference time matrix is compared with t_0^n. Any configuration of n_j^* with which the corresponding tail latency exceeds t_0^n is regarded as a latency violated configuration. We accept slight exceptions because outliers are inevitable. If the ratio of the latency violated batches is lower than the latency violation threshold τ_l, the exit is still acceptable.

For instance, as shown in Figure 7(c), when exit H_{e_1}/H_{e_2} is attached, the latency of some batches within the dataset are violated and the ratio of latency violated batches is not negligible. Thus, exit H_{e_1} and H_{e_2} are latency violated exits and should be filtered out first. Then the exit should be selected from the remaining exit candidates with the shortest average latency, which is H_{e_4}.

The searching target in posterior searching steps is different from that in searching for the first exit. In posterior steps, we estimate the tail latency instead of the average latency of the two-stage inference, and select the exit providing the shortest tail latency. That is because the batch size of the first stage is various in later steps and we can no longer estimate the average latency accurately. We estimate the tail latency by fixing the batch size of the first stage as the maximum allowed batch size of the previous step.

Suppose the last found exit is H_{e_l} and the maximum allowed batch size for stages after H_{e_l} is n_l, then the estimated two-stage tail latency after attaching H_{e_i} is $T_i^{\textbf{tail}} = t_{i,w/v} \approx T_{e_l \to e_i}^{n_l} + T_{e_i \to N}^{w}$, where w is the batch size of the second stage. Here, we omit the time consumed between stages because it is negligible compared to the execution time of engines. Thus, as long as $w < n_i$, where n_i is the maximum allowed batch size for stages after H_{e_i}, the tail latency violation is eliminated as proved in Equation 3.

$$T_i^{\textbf{tail}} = T_{e_l \to e_i}^{n_l} + T_{e_i \to N}^{w} < T_{e_l \to e_i}^{n_l} + T_{e_i \to N}^{n_i} < T_{e_l \to N}^{n_l} \qquad (3)$$

Thus, we pick the exit with the shortest estimated average latency in the first searching step and pick the exit with the shortest estimated tail latency in posterior steps.

There are two situations in which the searching for exits is terminated in advance. First, if none of the rest exit candidates can guarantee the task precision (the precision violated exits as shown in Figure 7(e)) or all exits violate the tail latency requirement, the samples will directly be delivered to the original final exit. Second, if all samples in all batches exit from a specific exit, we will also cease searching.

7 EVALUATION OF PAME

In this section, we evaluate the performance of PAME.

7.1 Implementation and Experimental Setup

In PAME, all the models are compiled with TensorRT [9] and the ONNX [7] format. During batched inference in PAME, in each inference stage, the corresponding model is first executed with TensorRT and the intermediate features of the queries are stored in an output buffer. Then, in the exit check phase, every single output value is analyzed by a thread in CUDA [41] and the instruction of data movement is then generated. Since the features of moving-on queries may not be contiguous in the memory space, we assign threads for each query to copy the features to the input buffer of the next stage. The copy incurs minor overhead as shown in Section 7.7.

Table 1 describes the experimental setup and lists 4 representative DNN tasks used to evaluate PAME in Computer Vision (CV) and Natural Language Processing (NLP) fields. The tasks cover the popular CNN and transformer backbones. We apply the widely-used ResNet101 as the backbone for CV tasks and BERT-base for NLP tasks. For semantic segmentation, we use a dilated variation of ResNet101 used in the original work [51]. We use *small*, *medium*, and *large* batch sizes for all the tasks to evaluate PAME. The batch sizes are $2, 4, 8$ for the semantic segmentation task, as the input batch has a size of $(bs, 3, 1024, 2048)$, which is sufficiently large. In this case, semantic segmentation can fully utilize the GPU with a small batch size. The three batch sizes are $16, 32, 64$ for other tasks.

Table 1: The hardware, software, and benchmarks.

Hardware	CPU: Intel Xeon Silver 4210, GPU: Nvidia Tesla V100			
OS & Driver	Ubuntu: 18.04.1 (kernel 5.4.0); GPU Driver: 470.57			
Software	CUDA: 11.4; TensorRT: 8.03; ONNX 11; Pytorch 1.9.1; Python 3.7.4			
Backbone	ResNet101 [27]			BERT-base [21]
Head	FC layer	Simple [48]	OCRNet [45]	FC layer
Benchmark	Imagenet [20], Imagenette; Imagewoof [5]	MPII human-pose dataset [10]	Cityscapes [15]	GLUE [44] (MRPC, SST-2, QNLI, RTE)
Task type	Classification	Pose estimation	Semantic segmentation	Classification

When training multi-exit models for the above DNN tasks, we empirically use 11 exit candidates for both ResNet101 and BERT-base. More candidates does not improve the results. The positions of exit candidates for ResNet101 backbones are $e_i = 3i - 2, i \in [1, 11]$, so the interval between exit candidates in ResNet101 is three blocks and the first exit candidate's position $e_1 = 1$. The positions of exit candidates for BERT-base backbone are $e_i = i, i \in [1, 11]$.

The baseline models are pre-trained models provided in released model repositories. We fine-tune pre-trained models to obtain multi-exit models. Unless otherwise stated, the task precision tolerance $\eta = 0.99$ and the latency violation threshold $\tau_l = 0.02$ in the recursive exit searching process. For each benchmark, we divide its validation dataset into two parts evenly without overlap. One part is used to simulate the inputs, with which PAME determines the to-be-used exits. The other part is used to evaluate the performance of the generated multi-exit model.

As the existing serving systems [6, 16] do not perceive the precision information during a batch's inference, they all use the single-exit inference. The algorithm works on proposing multi-exit models added many exits only for the high precision [30, 32, 54]. The added exits inevitably introduce heavy latency overhead for the queries that do not exit early. They often result in the long tail latency. To this end, we use the widely-used single-exit batch inference system, TensorRT [6], as the baseline.

7.2 Reducing the Response Latency

Figure 9 shows the average and the tail latencies of the tasks with the multi-exit models trained with PAME. In the figure, "S.", "M.", and "L." in the x-axis shows the case with the small, medium, and large batch sizes respectively. The horizontal dashed lines show the latency of the single-exit model with TensorRT. For a single-exit inference, the average latency and the tail latency are the same.

As observed, PAME reduces the response latency of all the tasks in all the cases. Compared with the single-exit inference, PAME reduces the average query latencies by 41% for three image classification tasks (*Imagenet*, *Imagenette*, *Imagewoof* in Figure 9(a)), 24% for pose estimation task (*MPII* in Figure 9(b)), 24.1% for semantic segmentation task (*Cityscapes* in Figure 9(c)) and 44.8% for four NLP tasks (*MRPC*, *QNLI*, *SST-2*, and *RTE* in Figure 9(d)) on average. The tail latencies are also reduced by 23.7%, 11.8%, 24.1%, and 37.2%, respectively. PAME can reduce the average tail latencies of the benchmarks, because it only ensures that the tail latency does not increase in the worst case and the average tail latency is usually shorter than the longest tail latency.

The latency reduction results from the early exits found by PAME. The found exits are shown in Figure 10. As we can see, only one

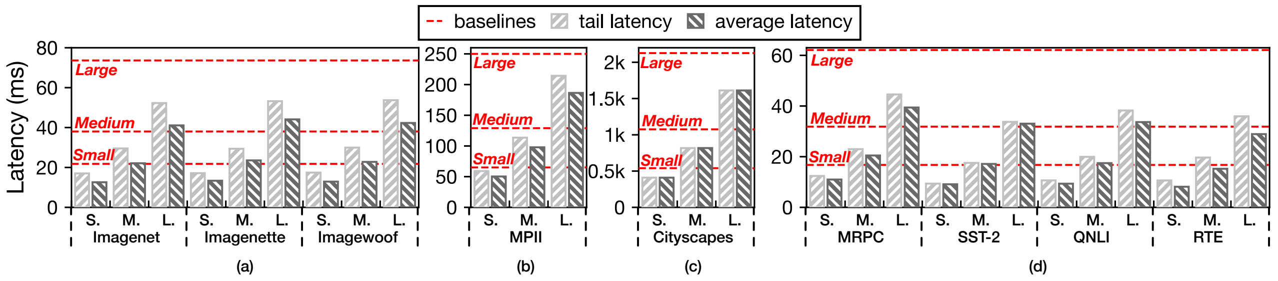

Figure 9: The tail and average latencies of batched queries in different tasks with PAME.

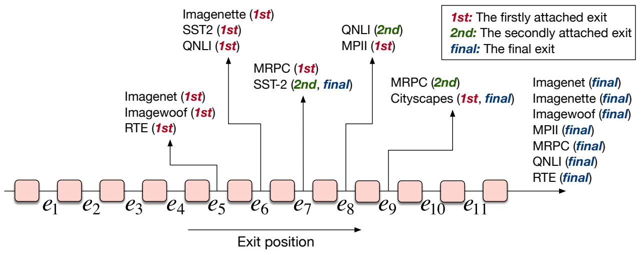

Figure 10: Exits found by PAME for the tasks.

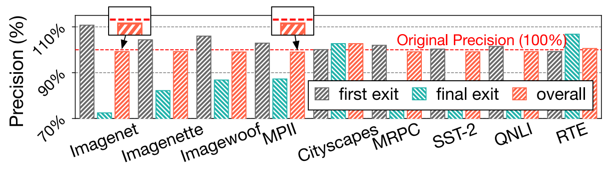

Figure 11: The normalized task precision at different exits.

early exit is found in *Imagenet*, *Imagenette*, *Imagewoof*, *MPII* and *RTE*. Two exits are found in *MRPC*, *QNLI* and *SST-2*. Many queries in a batch return from the early exits. We can also find that the tail latency and the average latency are the same in *Cityscapes* as shown in Figure 9. This is because all the queries in all batches actually return from the early exit found by PAME.

Figure 11 shows the average precision of the queries that return from the first exit, the average precision of queries that return from the final exit and the overall precision. The precisions are normalized to the original single-exit model's precision. The overall precision is 99.68% of the original model on average. Except for *Cityscapes* and *RTE*, the precisions of the first exit are already higher than baselines, which validates the effectiveness of early exiting; the precisions of the final exit are lower than baselines because the queries that arrive at the final exits are more difficult. Although slight task precision degradation is tolerated and expected, results on *Cityscapes* and *RTE* show better precisions (103% and 101% separately). This is because more effective exits are obtained through parameter fine-tuning in these two cases.

7.3 Effectiveness of the Exit Selection

In this experiment, we show whether PAME finds the appropriate exits from many exit candidates. We use *Imagenet* classification task as the benchmark in the subsection due to the limited space. Other tasks show similar results.

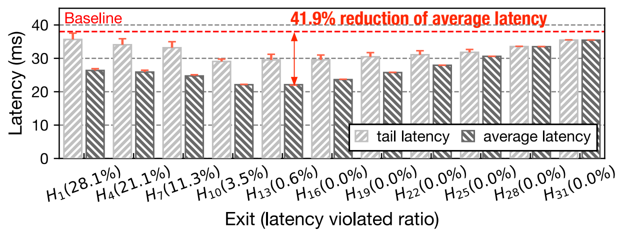

Figure 12: Latencies of queries when an exit is attached at different positions in the *Imagenet* classification task.

Figure 12 shows the latency reduction when the first exit is added at different positions. Let H_α represent an exit at the end of the block with block ID α. Then the exit candidates are $H_1, H_4, \cdots, H_{31}$, with the block ID of exit candidates being $e_i = 3i - 2$. It is worth noting that exits H_1, H_4, H_7 and H_{10} are not applicable actually because the corresponding real latency violated ratios exceed the latency violation threshold τ_l, which is 0.02 set in our experiment. The exit H_{13} has the lowest average latency among the remaining exit candidates, as the preceding exit candidates do not allow many samples to exit and posterior exit candidates are too late for reducing latencies. H_{13} is the fifth exit candidate for *Imagenet* classification as shown in Figure 10.

The error bar indicates the difference between the real latencies and the estimated latencies calculated from the inference time matrix and the batching pattern matrix. The estimated tail latency and the estimated average latency are accurate approximations, with less than 5% relative error. The latencies with different exits in other tasks share a similar pattern.

7.4 Effectiveness of the Inference Time Matrix

The inference time matrix and the batching pattern matrix determines the to-be-used exits in a task. In this experiment, we look into the inference time matrices of different tasks, and explain why different exits are found for them.

Figure 13 shows the inference time matrices of different tasks when searching for the first exit. As observed, the inference time matrices of ResNet101+FC and ResNet101+Simple [48] show high differences. The allowed batch size for the second stage after attaching FC-layer-based exits is not significantly reduced, because the FC layer itself costs little. However, due to the large computation brought by the deconvolution layers in Simple [48], the allowed batch sizes for the second stage are rather low no matter where the exit is attached (Figure 13(b)).

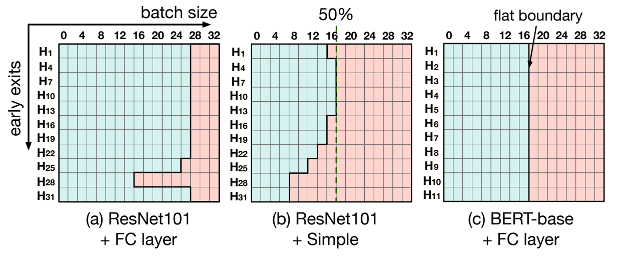

Figure 13: Inference time matrix in different tasks. Batch sizes in the green region are allowed for the second stage.

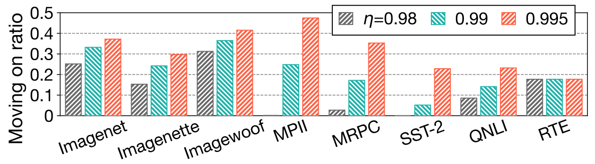

Figure 14: The moving-on ratio after the first exit with different precision tolerances.

We can also find that there is a flat boundary in the inference time matrix for Bert-based tasks in Figure 13(c). The flat boundary originates from the compiler mechanism. In general, the compiler performs padding for the batch. For example, the elapsed time of running an inference with batch size 9−15 is similar to the inference with batch size 16.

Some readers may find that the last early exit (H_{31}) in Figure 13(a) allows larger batch sizes for the second stage inference than the penultimate exit (H_{28}). This is because the exit structures of them are different (H_{28} includes an additional feature alignment block while H_{31} only includes the FC head). The large computation overhead in H_{28} suppresses the batch size allowed for the second stage.

7.5 Effect of Adding Multiple Exits

Figure 11 shows that with the first exit attached, the overall precision has already been reduced in most cases. In general, if the overall precision has already reached the precision tolerance, it is not necessary to add more exits.

Figure 14 shows the moving-on ratio of queries after the first exit. As observed, the moving-on ratio of samples is only 21.6% on average when $\eta = 0.99$. If we allow more tolerance of the precision degradation by setting $\eta = 0.98$, the moving-on ratio is 12.6% on average. The moving-on ratio even becomes zero (all samples exit from the first exit) for *MPII* and *SST-2*. If we set the precision tolerance to 0.995, there are still more than half of the samples would exit from the first exit. Therefore, if we apply exits after the first exit, there are only opportunities that the latencies of less than half of the samples can be reduced.

Among tasks with multiple early exits, we only show the latencies with different numbers of exits on the task of *QNLI* in Figure 15. Other tasks with multiple early exits (*SST-2, MRPC*) share a very similar pattern with *QNLI*. We record the average latency in the 1-exit configuration with exit H_6 and the 2-exit configuration with exit H_6 and H_8. Compared with the 1-exit configuration, neither the tail latency nor the average latency is reduced dramatically (reduced by 5% for the tail latency and 0.8% for the average latency) with the 2-exit configuration. Generally, the latency reduction is gradually reduced with the increasing of the exit count.

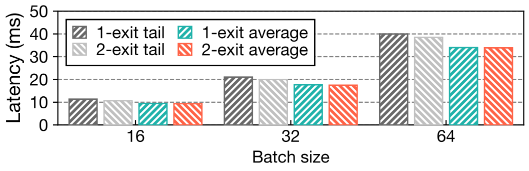

Figure 15: Latencies of *QNLI* with different numbers of exits.

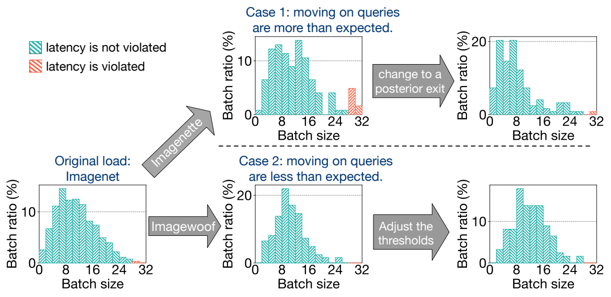

Figure 16: The effectiveness of adapting to the load changes.

7.6 Adapting to Load Changes

In a specific period of serving, the input data distribution may be different from the distribution of data based on which the inference system is initialized. We evaluate PAME in adapting to the load changes in this subsection. In this experiment, we use ResNet101+FC as the model for the image classification task. Specifically, we train the multi-exit model with *Imagenet*, and use *Imagenette* [5] and *Imagewoof* [5], to be the validation dataset. Other tasks show similar results. Figure 16 shows the cases when the input data pattern changes from *Imagenet* to *Imagenette* or *Imagewoof.*

If the input load changes to *Imagenette* (case 1), the moving-on queries at the exit will be more than expected. The latency violation ratio will be increased from 0.003 to 0.065. In this case, we have to move the exit to a posterior position where the latency violation is bearable. With *Imagenette*, the new exit is H_{16}. After that, the latency violation ratio is reduced to 0.008.

If the input load changes to *Imagewoof* (case 2), the moving on queries will be less than expected. The latency violation ratio is decreased to 0, but the normalized task precision reduces from 99.1% to 97.8%. In this situation, the thresholds in exit check structures should be adjusted first to meet the precision requirement. After raising the confidence threshold from 0.55 to 0.6, the task precision is increased to 99.2%.

Incremental learning can be used to quickly update the multi-exit model based on the up-to-date input pattern. Once the input pattern change is observed, the adjustment can be done based on the inference time matrix and batching pattern matrix directly.

In general, if frequent latency violation is encountered, PAME moves the exit to a posterior position of the backbone instead of adjusting the exit check structure, since changing the moving-on ratio brings great risk of degrading the task precision. If the precision is not guaranteed, PAME raises the threshold of current exit to improve the task precision.

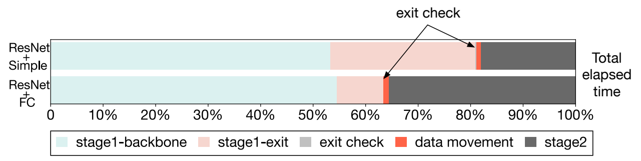

Figure 17: The tail latency breakdown.

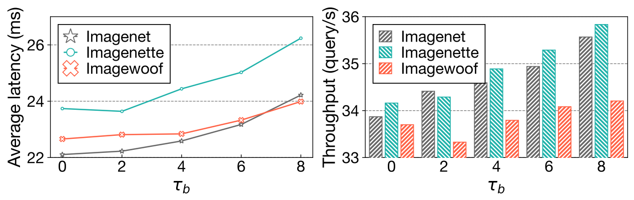

Figure 18: The latency and the throughput of image classification tasks with PAME and parallel batch scheduling.

7.7 Overhead Analysis

The attached exits as well as the exit check and the data movement between stages may incur runtime overhead. Figure 17 shows the average latency breakdown of queries that do not return early in the two-stage inference. Due to limited space, we only show two typical tasks: *Imagenet* on ResNet101+FC for image classification, and *MPII* on ResNet101+Simple for pose estimation. Other tasks show similar trends.

As observed, the main overhead comes from the computation of exits. For *Imagenet*, the exit computation in stage 1 occupies 8.9% of the total model computation time. For *MPII*, the exit computation occupies 27.9% of the total time. Overall, the execution of models takes up to 98.9% of the total inference time. The exit check phase only occupies 0.1% of the total inference time and the data movement occupies 1.0% on average. Thus, the overhead from exit check and data movement is insignificant compared to the computation of exits.

7.8 Opportunity of Parallel Batch Scheduling

If the hardware is not fully utilized when the batch size is small in later stages of a batch, parallel batch scheduling introduced in Section 4.2 can be used to increase the serving throughput. Whenever the number of queries being processed is lower than the batch size threshold τ_b, a new batch can be enqueued to the engine. By adjusting τ_b, we can have a trade-off between the average latency and the overall throughput.

Take image classification tasks as examples. As shown in Figure 18, both the throughput and the average latency of image classification tasks increase with τ_b. However, the throughput improvement is limited because large τ_b will cause frequent violation in tail latency. When $\tau_b \leq 4$, the ratio of latency-violated samples is below the latency violation threshold $\tau_l = 0.02$. The ratio of latency-violated samples becomes 0.03 when $\tau_b = 6$ and 0.12 when $\tau_b = 8$, which are both unacceptable.

PAME improves 28.9% throughput in image classification tasks on average, and the throughput improvement increases to 31.3% with the parallel batch scheduling.

8 CONCLUSION

In this work, we propose a precision-aware mechanism to attach multiple exits on DNN models to reduce latencies. The proposed multi-exit DNN serving system allows queries to run in batch but exit independently. First, we determine the exit structure by designing bridges, heads and exit check structures separately. Second, we apply a method to determine the to-be-used exits based on input patterns. PAME takes the exit overhead into consideration and the tail latency of a batch is not increased. Overall, PAME achieves up to 51.4% latency reduction while maintaining 99.68% precision of original models on average.

ACKNOWLEDGEMENTS

We would like to thank Prof. Christos Antonopoulos for his suggestions on this work during the shepherding phase. This work is partially sponsored by the National Natural Science Foundation of China (NSFC) (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

REFERENCES

[1] Amazon rekognition. https://aws.amazon.com/rekognition/.
[2] Amazon translate. https://aws.amazon.com/translate/.
[3] Google translate. https://translate.google.com.
[4] Huggingface pre-trained models. https://huggingface.co/transformers/v3.3.1/pretrained_models.html.
[5] Imagenette and imagewoof. https://github.com/fastai/imagenette.
[6] Nvidia triton inference server. https://github.com/NVIDIA/triton-inference-server.
[7] Onnx. https://github.com/onnx/onnx.
[8] Pytorch models and pre-trained weights. https://pytorch.org/vision/stable/models.html.
[9] Tensorrt. https://developer.nvidia.com/tensorrt, 2021.
[10] Mykhaylo Andriluka, Leonid Pishchulin, Peter Gehler, and Bernt Schiele. 2d human pose estimation: New benchmark and state of the art analysis. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, pages 3686–3693, 2014.
[11] Arian Bakhtiarnia, Qi Zhang, and Alexandros Iosifidis. Multi-exit vision transformer for dynamic inference. *arXiv preprint arXiv:2106.15183*, 2021.
[12] 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.
[13] Kwan Ho Ryan Chan, Yaodong Yu, Chong You, Haozhi Qi, John Wright, and Yi Ma. Redunet: A white-box deep network from the principle of maximizing rate reduction. *arXiv preprint arXiv:2105.10446*, 2021.
[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 578–594, 2018.
[15] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.
[16] 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*, pages 613–627, 2017.
[17] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Minyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design*, pages 497–505. IEEE, 2019.
[18] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[19] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[21] 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.

[22] Rahul Duggal, Scott Freitas, Sunny Dhamnani, Duen Horng Chau, and Jimeng Sun. Elf: An early-exiting framework for long-tailed classification. *arXiv preprint arXiv:2006.11979*, 2020.

[23] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision. In *2020 IEEE/ACM Symposium on Edge Computing*, pages 84–95. IEEE, 2020.

[24] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[25] Yue Guan, Zhengyi Li, Jingwen Leng, Zhouhan Lin, Minyi Guo, and Yuhao Zhu. Block-skim: Efficient question answering for transformer. *arXiv preprint arXiv:2112.08560*, 2021.

[26] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. Federated visual classification with real-world data distribution. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part X 16*, pages 76–92. Springer, 2020.

[29] Ting-Kuei Hu, Tianlong Chen, Haotao Wang, and Zhangyang Wang. Triple wins: Boosting accuracy, robustness and efficiency together by enabling input-adaptive inference. In *International Conference on Learning Representations*, 2019.

[30] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. Multi-scale dense networks for resource efficient image classification. In *International Conference on Learning Representations*, 2018.

[31] Weiyu Ju, Wei Bao, Liming Ge, and Dong Yuan. Dynamic early exit scheduling for deep neural network inference through contextual bandits. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 823–832, 2021.

[32] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*, pages 3301–3310. PMLR, 2019.

[33] Alexandros Kouris, Stylianos I Venieris, Stefanos Laskaridis, and Nicholas D Lane. Multi-exit semantic segmentation networks. *arXiv preprint arXiv:2106.03527*, 2021.

[34] Stefanos Laskaridis, Stylianos I Venieris, Hyeji Kim, and Nicholas D Lane. Hapi: hardware-aware progressive inference. In *2020 IEEE/ACM International Conference On Computer Aided Design*, pages 1–9. IEEE, 2020.

[35] Hao Li, Hong Zhang, Xiaojuan Qi, Ruigang Yang, and Gao Huang. Improved techniques for training adaptive deep networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1891–1900, 2019.

[36] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

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

[38] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[39] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[40] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[41] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[42] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[43] 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*, pages 2464–2469. IEEE, 2016.

[44] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[45] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, et al. Deep high-resolution representation learning for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[46] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision*, pages 409–424, 2018.

[47] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8817–8826, 2018.

[48] Bin Xiao, Haiping Wu, and Yichen Wei. Simple baselines for human pose estimation and tracking. In *Proceedings of the European conference on computer vision*, pages 466–481, 2018.

[49] 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.

[50] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P Xing. Cavs: An efficient runtime system for dynamic neural networks. In *2018 USENIX Annual Technical Conference*, pages 937–950, 2018.

[51] Yuhui Yuan, Xilin Chen, and Jingdong Wang. Object-contextual representations for semantic segmentation. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VI 16*, pages 173–190. Springer, 2020.

[52] Shulai Zhang, Zirui Li, Quan Chen, Wenli Zheng, Jingwen Leng, and Minyi Guo. Dubhe: Towards data unbiasedness with homomorphic encryption in federated learning client selection. In *50th International Conference on Parallel Processing*, pages 1–10, 2021.

[53] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–373, 2022.

[54] 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.