POLITECNICO DI TORINO
Repository ISTITUZIONALE

ADEx: Adaptive Early Exit DNNs for Inference Robustness to Popularity Drift

Original
ADEx: Adaptive Early Exit DNNs for Inference Robustness to Popularity Drift / Zhao, Z., Brevi, D., Pastrone, C., Levorato, M., Malandrino, F., Chiasserini, C.F.. - (2026). (2026 IFIP Networking Conference (IFIP Networking) Lugano (Swi) 24 - 27 May 2026).

Availability:
This version is available at: 11583/3009727 since: 2026-04-09T11:49:45Z

Publisher:
IFIP

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

14 September 2026

ADEx: Adaptive Early Exit DNNs for Inference Robustness to Popularity Drift

***Abstract*—Deep neural networks (DNNs) with Early Exits (EE) are widely adopted at the network edge to reduce inference latency and resource consumption by allowing samples to exit the DNN early when sufficient confidence is achieved. However, their effectiveness significantly degrades under *popularity drift* – where the frequency of target classes shifts over time. This paper introduces ADEx, an Adaptive Drift-aware EE framework that enables dynamic and lightweight adaptation of DNNs with EEs to class popularity drift. ADEx continuously monitors EE behavior to detect popularity shifts and selectively retrains only the affected ones, while keeping the backbone network unchanged. For unsupervised adaptation, ADEx uses the final exit as a teacher to pseudo-label new data, applying a priority-aware loss that enforces high confidence on popular classes and uncertainty on others. Furthermore, we analyze execution strategies balancing adaptation speed, latency, and resource usage. Experimental results demonstrate that ADEx restores inference efficiency, reducing main-branch usage from a post-drift peak of 46.14% to 7.65% (near the 7.46% pre-drift level) and lowering mean latency from 3.05 ms to 2.57 ms. Compared to a Joint Fine-tuning baseline, ADEx achieves similar accuracy recovery while reducing peak GPU memory by 52.3% and avoiding full-model shadow copies.**

***Index Terms*—Edge computing, Dynamic DNNs, Early Exit architectures, Popularity drift**

I. Introduction

Deep Neural Networks (DNNs) are increasingly deployed at the network edge to enable intelligent applications with low latency and enhanced privacy preservation, However, this paradigm faces two fundamental and interconnected challenges. *First*, edge devices (e.g., IoT sensors, mobile user devices) are severely constrained by computational power, memory, and battery life, creating a critical trade-off between model performance and resource consumption. *Second*, real-world data are often non-stationary: the statistical properties of data streams evolve over time due to changing environments or user behaviors. These challenges are not orthogonal. The efficiency of many edge systems deploying machine learning (ML) models on edge nodes rely on a stable data distribution. Thus, a model optimized to efficiently process "common cases" will suffer significant performance and efficiency degradation when the definition of a "common case" changes.

In our work, we focus on classification tasks (which can be performed via DNNs such as ResNet) [1], and address the case where class probability distribution may vary over time. More formally, let $\mathcal{C}$ be the set of classes, and $p_c(t)$ denote the prior probability that a newly-arrived sample x belongs to class $c \in \mathcal{C}$. When a drift occurs, the class distribution shifts over time, i.e., there exists a time interval T such that $p_c(t) \neq p_c(t+T)$. Hereinafter, we refer to the above drift in the output distribution as *popularity drift*. Unlike traditional concept drift, in popularity drift the set of classes $\mathcal{C}$ remains constant while their relative frequencies change.

Current solutions fail to address the above challenge comprehensively. Indeed, traditional computation offloading frameworks have focused on finding the optimal static split-point for a fixed DNN to balance latency and energy [2]–[4]. Recent works, such as [5], have further refined this by optimizing partitions based on available resources *at training time*. While effective for a fixed environment, these popularity drift-unaware methods cannot adapt to post-deployment shifts in the data distribution. Other works leverage Early-Exit (EE) DNN architectures for *sample-level* adaptation (i.e., easy- vs. hard-to-infere samples) [6], but their adaptation logic (e.g., confidence thresholds) is static and fixed once training is over. These approaches are vulnerable to efficiency collapse when the underlying data distribution (e.g., the definition of an "easy" class) changes. As for traditional popularity drift adaptation techniques, they often propose monolithic retraining of the entire model [7], [8]. These resource-heavy approaches are fundamentally ill-suited for inference execution at the edge. A full model retraining incurs prohibitive costs in on-device computation, energy, and data transmission, making frequent updates impractical [9].

Our approach. To bridge the above critical gaps, we focus on EE architectures and propose the *Adaptive Drift-aware Exits (ADEx)* framework. Our first key insight in designing ADEx is that, in dynamic DNNs with EEs, the association between EEs and input samples should be driven by the popularity of different groups of samples. In other words, the samples of the most common groups should be handled by the earliest exists so that the global inference latency and cost can be minimized. However, if such a process is implemented by applying a strict specialization on the EEs, i.e., by training them to statically process only the "common cases" or by reacting to drift with a costly, monolithic update, we may end up obtaining a highly inefficient system. To overcome this issue, ADEx adopts an unsupervised popularity drift detection mechanism and, whenever deemed necessary, triggers a lightweight, targeted retraining to re-tailor the affected EE branches on the newly detected "common cases".

Furthermore, unlike existing works (e.g., [10]), which use DNN architectures with EEs for one-time domain adaptation through techniques like knowledge distillation, ADEx dynamically and automatically adapts to drift through a complete, closed-loop system of popularity drift detection, sample pseudo-labeling, and EE update. The loop uses the high-

accuracy main branch as a "teacher" to generate pseudo-labels for the new data samples that cannot be accurately processed through EEs, enabling unsupervised adaptation. Importantly, ADEx enables such an adaptation to restore inference quality *without altering the physical deployment configuration*: an exit branch on a device can be updated in-place, eliminating the operational complexity of redeployment. Additionally, as demonstrated in Sec. V, through ADEx, the DNN model can recover from performance degradation due to popularity drift by updating only a small fraction of its parameters.

Our contributions. Our key contributions are as follows:

- An adaptive framework, named ADEx, that trains the EEs of dynamic DNNs at the network edge so that the most common types of data samples are processed by the earliest EEs, thus decreasing the cost and latency of edge inference. Furthermore, it does so by integrating unsupervised popularity drift detection with *exit-specific retraining*, which makes the system capable to dynamically adapt to shifts in both the popularity and the distribution of the data samples.
- A flexible data samples pseudo-label strategy using the main branch as a "teacher" to enable rapid, unsupervised adaptation of EEs without new ground-truth labels.
- A comprehensive analysis of different dynamic EEs retraining strategies and their trade-offs in terms of performance recovery, resource cost, and service availability.
- An extensive experimental study showing that ADEx recovers most of the accuracy lost due to popularity drift within a short adaptation window, while requiring only a small fraction of the model parameters to be updated, and incurring negligible disruption to online inference. Specifically, ADEx improves accuracy to 92.52% (from 91.20% post-drift), and restores EE efficiency by reversing the efficiency collapse caused by the drift—reducing main-branch usage from 46.14% (post-drift, pre-adaptation) to 7.65% (after ADEx adaptation). This outperforms state-of-the-art test-time adaptation (TTA) baselines like TENT and EATA, which, despite buffering, fail to re-specialize EEs and leave the system in a backbone-heavy regime (e.g., TENT-buffered maintains 44.22% main-branch usage). While the Joint FT baseline provides a faster updating time, it incurs prohibitive system costs, requiring 2.1× more GPU memory and a full model duplication to maintain service continuity. In contrast, ADEx's targeted adaptation achieves near-optimal performance with significantly lower resource consumption.

The rest of the paper is organized as follows. Sec. II reviews some relevant related work and provides some background on dynamic DNNs. Sections III and IV present, respectively, our system design and the ADEx framework. Sec. V introduces our experimental methodology and compares ADEx against existing solutions. Finally, Sec. VI draws our conclusions and possible future research directions.

II. Related Work and Background

Our work is related to two primary research areas: DNN execution at the edge, and ML model adaptation to popularity drift and Out-Of-Distribution (OOD) detection.

DNNs at the Edge. Research on efficient DNN execution on edge servers and edge devices has mainly focused on optimizing static DNN models. This is achieved via two main strategies: (i) by optimizing the model architecture itself (e.g., model compression, dynamic networks, and lightweight model design), and (ii) by optimizing the deployment and execution environment (e.g., computation partitioning and offloading) [2], [3]. Among the works exploiting dynamic architectures, several paradigms exist to adapt computation to input difficulty, aiming to allocate resources proportionally to the complexity of each sample. These include *dynamic width* approaches like Slimmable Networks [11] and Adaptive Width Networks [12] that adjust channel numbers at runtime, as well as *dynamic routing* mechanisms such as [13] and [14] that selectively skip layers or blocks. Our work follows the *dynamic depth* paradigm established by BranchyNet [15], which leverages EEs to adapt the way input is processed. We adopt these modular EE DNN architectures as well; *however, we do not limit inference to static DNN execution, but we allow for continuous adaptation of the DNN by triggering EE retraining whenever needed.*

Another line of work optimizes the deployment of a DNN by splitting it across the device-edge boundary [3], [16]. The goal is to find an optimal partition point in the DNN architecture, which balances on-device computation with the latency and energy cost of offloading, also exploiting dynamic split [4], collaborative approaches [17], [18] and TPUs [19].

However these methods are inherently popularity drift-unaware, which could make their calculated optimal partition sub-optimal. *In contrast, ADEx is designed specifically for a dynamic scenario, enabling lightweight model adaptation without requiring a costly and complex re-partitioning or re-deployment of the whole DNN.*

Concept Drift/OOD Detection and Model Adaptation. The phenomenon of popularity drift addressed by ADEx is linked to concept drift, i.e., non-stationary data as the DNN input. Concept drift is well-studied [7], and a variety of methods exist to detect when a concept drift has occurred. Classic methods like DDM and EDDM monitor model performance metrics (e.g., error rate) [20]. ADEx leverages these advanced detection techniques as the trigger for its adaptation phase.

Most traditional drift adaptation strategies respond to a detected drift by initiating a monolithic retraining of the entire model, either from scratch or by fine-tuning on a new window of data [7], [21]. While effective, these resource-heavy approaches are fundamentally ill-suited for the edge, as they incur prohibitive costs in computation, energy, and time [22], [23]. Our work fills this gap: instead of this all-or-nothing retraining, we introduce a lightweight, surgical adaptation. By leveraging the modularity of EE architectures, we update only the EE branches affected by the current drift, which represent a small fraction of the total model parameters. This targeted approach, combined with our teacher-student pseudo-labeling strategy [24], yields a swift and resource-efficient adaptation, which makes it feasible to execute DNNs at the network edge.

Regarding EE training [25], [26], the seminal work in [15]

has provided the foundation of EEs through the BranchyNet solution. Subsequent works, such as ClassyNet [27], employ an explicit rejection strategy: an additional output class (e.g., "rejected" or "other") is explicitly added to the classifier to absorb difficult or non-target samples (a.k.a. OOD samples). While effective, explicit rejection requires modifications to the network architecture (changing the output layer dimension) and often maintaining a balanced set of "negative" samples during training to train the extra class. In contrast, detailed later, by training the exit to output a uniform distribution for non-target classes, we allow the system to use simple entropy-based thresholds for exiting decisions without any structural changes. *This local specialization of EEs is crucial for ADEx, as it allows us to dynamically redefine which classes should be targeted by an EE target and which should not, without retraining the final layer's weights or the model architecture*.

Recent works such as RACENet [28] have also proposed adaptive schemes that use class-aware weights to prioritize specific classes at edge exits. While RACENet effectively directs the model's attention to target classes via weighted cross-entropy, it lacks a mechanism to explicitly suppress the confidence of non-target classes. However, an exit specialized for common cases might still yield high-confidence predictions for rare cases simply because they share some features, leading to overconfident misclassifications (false early exits). Our ADEx overcomes this issue by explicitly penalizing low-entropy distributions for non-target classes. This ensures that, under ADEx, rare samples do not just fail to match the target classes, but are actively forced into a high-uncertainty state, making them easy to detect and defer to deeper layers.

III. System Design and Dynamic DNN Model

We consider a network system where mobile devices, constrained by limited computational and energy capabilities, offload inference tasks to an edge server. As illustrated in Fig. 1, the edge server employs a multi-exit DNN architecture to execute these tasks *timely and efficiently*. By integrating multiple classifiers at intermediate layers, the architecture allows for early results based on confidence, significantly reducing resource consumption. Within this edge-based inference system, we introduce the Adaptive Drift-aware Exits (ADEx). As detailed below, ADEx operates on the edge server to adapt the multi-exit DNN to changing data streams.

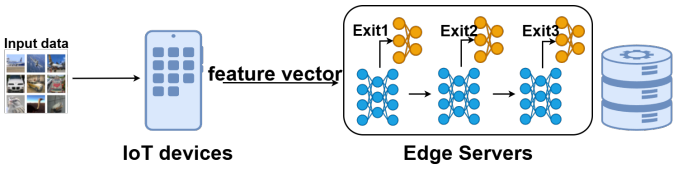

Fig. 1: Inference task offloading from a device to an edge server where the task is executed via a DNN with early exits.

The system architecture we consider for ADEx (Fig. 2) includes two main elements, i.e., a DNN with EEs (EEs) and a decision-making entity called *EE Update Controller*.

DNN with EEs. The DNN is composed of a sequence $k_1, \ldots, k_K \in \mathcal{K}$ of blocks. At the end of every block k, there is an EE, i.e., a *classifier*, which, given the block's output $f^k(x)$, computes the probabilities $\hat{y}_c^k(x)$ that sample x belongs to class $c \in \mathcal{C}$. If the probability $\hat{y}_{c^\star}^k(x)$, associated with the most probable class $c^\star$, is higher than a threshold τ, then the DNN returns $c^\star$ as the classification decision, and the sample is not processed further. Processing a sample at each block k incurs a cost γ^k and a latency λ^k. To keep both low, it is important to exit the DNN as early as possible, for as many samples as possible.

As mentioned, popularity drift differs from traditional concept drift since in popularity drift the set $\mathcal{C}$ of classes itself does not change. This also implies that all samples are classified eventually, even if (in the most unlucky cases) they may have to traverse the whole DNN. By indicating with $\pi_c^{k,t}$ the probability that a sample of class c arriving at time t is classified by EE k, then we have: $\sum_{k\in\mathcal{K}} \pi_c^k(t) = 1, \forall c \in \mathcal{C}, t \in \mathcal{T}$. We also remark that popularity drift in our case corresponds to the detection of concept drift/OOD locally at the single EEs.

EE Update Controller. The controller is in charge of:

- Monitoring the output $f^k(x)$ of the K blocks, checking for *local* popularity drifts;
- If such a popularity drift is detected, triggering a retrain of the affected EE(s);
- Setting the loss function used for retraining, so as to specialize the EE(s) to different classes;
- Storing and providing additional samples $\mathcal{X}'$ to speed up the next EE updating (i.e., retraining) process.

The controller has a global view of the learning task at hand, i.e., it is aware of the outputs $f^k(x)$ of all blocks, and of all probabilities $\hat{y}_c^k(x)$ predicted by EEs. As detailed in Sec. IV, it can use such information to make local decisions about which EE(s) to retrain, when, and using which data.

IV. The ADEx Algorithmic Suite

To make the above decisions, the controller executes the ADEx algorithmic suite. The key idea behind ADEx is that the best way to run a DNN with EEs is to ensure that:

(i) EEs are *specialized*, i.e., each EE only seeks to classify a subset of all classes $\mathcal{C}$;
(ii) Specialization is *popularity driven*, i.e., earlier EEs are specialized in the most popular classes.

Fig. 3 depicts how different exits are optimized to prioritize specific class popularity. To ensure that these goals are met, the controller has to perform the already mentioned four tasks, as detailed below.

A. Popularity drift detection and EE retraining triggering

Checking for popularity drift. Popularity drift detection coincides in our case with local concept drift/OOD detection at the single EEs. Accordingly, ADEx implements a lightweight **frequency-based monitor** that operates directly on the resulting classification counts. The detailed procedure for this monitoring is formalized in Algorithm 1. This monitor tracks the empirical class distribution within a sliding window of recent inference results. Specifically, it monitors the cumulative frequency of samples that the EE identifies as belonging to

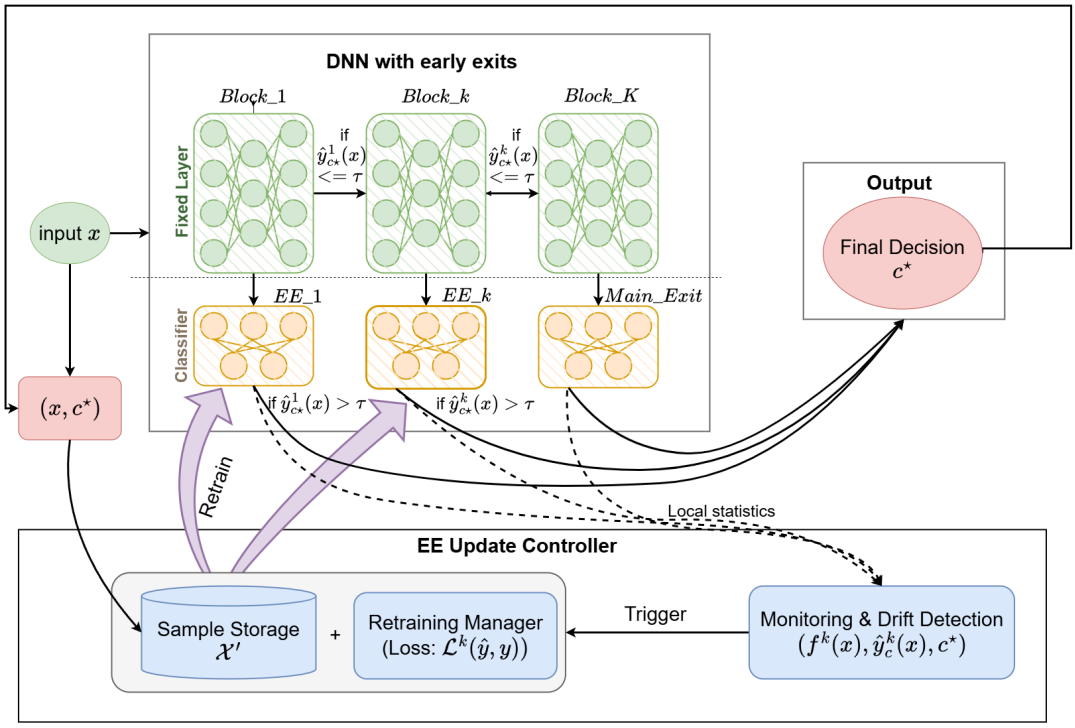

Fig. 2: ADEx's architectural overview, including a DNN pipeline with EEs and an EE Update Controller. Each block k is followed by a classifier (EE) that enables sample exit if the prediction probability exceeds threshold τ. The controller monitors local outputs $f^k(x)$ and $\hat{y}_c^k(x)$ to detect popularity drifts, subsequently triggering EE retraining using specialized loss functions and stored samples $\mathcal{X}'$.

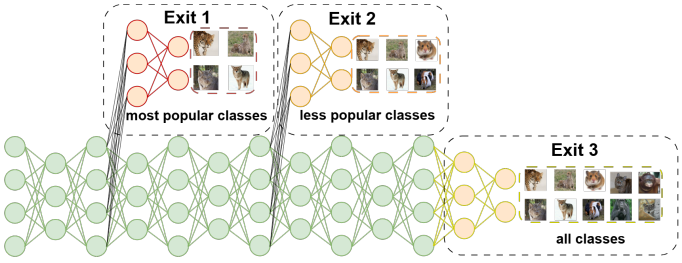

Fig. 3: EE specialization for different class popularity: Earlier exits (e.g., Exit 1) are optimized to efficiently classify the most popular classes, while deeper exits (e.g., Exit 2) handle less popular classes. The final exit (Exit 3) processes all classes that remain unclassified by previous exits, which typically includes more complex or rare instances, ensuring full coverage.

classes outside its current priority set $\mathcal{P}^k$, and the Statistical Divergence as the Jaccard-like discrepancy between $\mathcal{P}^k$ and the current top-N most frequent classes (see Alg. 1). A drift is triggered when the statistical divergence between the observed class distribution and the specialized profile of the EE exceeds a predefined threshold, or when the "miss rate" (samples deferred to deeper layers) significantly increases, signaling that the current specialization of the EE is no longer aligned with the incoming "common cases."

Data collection buffer. Retraining EEs requires training samples which *(i)* are labeled, and *(ii)* reflect the current, i.e., post-drift, class popularity. In practical cases, those requirements translate into high costs and/or long delays. To cope with this issue, ADEx includes a *data collection buffer* mechanism, operating as follows:

- The controller collects input samples into a new dataset $\mathcal{X}_{\text{new}}$;

Algorithm 1: Local Popularity Drift Detection, EE k

Input : Predicted class $c^\star$ for sample x at EE k
Parameters: Priority set $\mathcal{P}^k$, window size W, thresholds θ_{div} and θ_{miss}
Output : Retraining trigger $T \in \{\text{True}, \text{False}\}$

1. Update local inference history Append $c^\star$ to the sliding window B^k **if** $|B^k| > W$ **then**
  Remove the oldest entry from B^k

2. Evaluate specialization efficiency
$N_{\text{out}} \leftarrow \left|\{\, c \in B^k \mid c \notin \mathcal{P}^k \,\}\right|$; `// Samples outside the priority set`
$R_{\text{miss}} \leftarrow \frac{N_{\text{out}}}{W}$; `// Measure of usage decline`

3. Measure divergence from common cases ;
`// Find the current top-`$|\mathcal{P}^k|$ `most frequent classes in the window`
$\mathcal{C}_{\text{recent}} \leftarrow \text{TopFreq}(B^k, |\mathcal{P}^k|)$;
$\Delta \leftarrow 1 - \frac{|\mathcal{C}_{\text{recent}} \cap \mathcal{P}^k|}{|\mathcal{P}^k|}$; `// Fraction of recent common cases not covered by` $\mathcal{P}^k$

4. Triggering logic if $R_{\text{miss}} > \theta_{\text{miss}}$ ***or*** $\Delta > \theta_{\text{div}}$ **then**
  return True; `// Reassessment of` $\mathcal{P}^k$
return False

• If such samples turned out to be processed through a later exit than the intended one, then they are processed by the whole main branch. In this way, they are associated with pseudo-labels in $\tilde{\mathcal{Y}}_{\text{new}}$ obtained through the output of the last block K, *in lieu* of ground-truth labels, which are not readily available. Such pseudo-labels also included in $\mathcal{X}_{\text{new}}$. Instead, samples that exit at the intended EE are removed from $\mathcal{X}_{\text{new}}$.

• The samples in $\mathcal{X}_{\text{new}}$ are used for EE retraining.

The data collection buffer mechanism exploits the fact that, in a popularity drift scenario, the global DNN still produces reliable outputs, even if the outputs of individual EEs are affected by the drift. Furthermore, thanks to its ability to detect drifts earlier and to collect information from all sections of the DNN, the EE update controller is in an ideal position to collect and enhance the data collection buffer.

Triggering the retraining of EEs. Recall that each EE in the DNN handles a set of classes, called *priority classes*. Specifically, at time t, each EE $k \in \mathcal{K}$ handles the classes in set $\mathcal{P}^k(t) \subseteq \mathcal{C}$, whose size $P^k(t)$ is fixed for each EE, and known to the controller. Once a popularity drift is detected, the controller reacts to it by first reassessing and adjusting, if needed, the set of priority classes $\mathcal{P}^k(t)$ at each EE, so that the EE handles samples corresponding to the most popular classes. Then it retrains any of the EEs so that they can efficiently and reliably detect the samples in their priority classes. In more detail, each EE should be able to detect the n^k classes it is most likely to see. This should account for the probabilities that samples are classified at EEs earlier than k itself, i.e.,

$$\mathcal{P}^k(t) \leftarrow \arg \mathbf{top}^{\mathbf{N^k}}_{\mathbf{c}\in\mathcal{C}} p_c(t) \prod_{h<k} (1 - \pi^h_c(t)).$$

B. Priority-aware loss function

The main goal of retraining an EE in ADEx is restoring (or enforcing) its *specialization*. I.e., given EE $k \in \mathcal{K}$, we want to *(i)* classify with high accuracy the samples in its set $\mathcal{P}^k(t)$ of priority classes, while *(ii)* expressing uncertainty for the others. To pursue this goal, the loss incurred for each sample (x, y) at each EE k is defined as:

$$\mathcal{L}^k(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}_{\text{CE}}(\mathbf{p}^k \hat{\mathbf{y}}, y) + \mathcal{L}_{\text{KL}}((\mathbf{1} - \mathbf{p}^k)\hat{\mathbf{y}} || \mathcal{U}), \quad (1)$$

where we dropped time indices and denoted with $\hat{\mathbf{y}}{=}(\hat{y}^k_c(x))_c$, the vector of probabilities predicted by EE k, with y the label of the current sample x, and with $\mathbf{p}{=}(\mathbb{1}_{[c\in\mathcal{P}^k]})_c$, a binary vector indicating whether each class $c \in \mathcal{C}$ is a priority class for c. Then, the global loss is simply the sum of all local ones. The first term of (1) is the standard categorical cross-entropy, computed only with reference to the priority classes of EE k. For the other classes, the second term represents the Kullback-Leibler (KL) divergence, expressing how different the prediction of EE k for classes *other than* its priority classes is from uniform probabilities. The second term in (1) is one of the main innovation of ADEx, and represents a *local* and *conditional* regularization of each EE's outputs. We can see that term as pushing its EE to treat samples not in its priority classes as locally-OOD; in other words, none of the EE's classification power is wasted on samples that would be processed by later blocks anyway.

C. Adaptive retraining strategies

Once built the adaptation dataset $\mathcal{X}_{\text{new}}$, the system must update the EEs to restore their inference performance. However, retraining EEs consumes the same type of resources, e.g., CPU/GPU and memory, required for inference, hence, such an operation might interact with the latency constraint of the edge applications. Thus, the controller must select an execution strategy that best matches the available resources and quality of service (QoS) requirements. We identify a set $\mathcal{S}$ of three possible strategies, and make strategy selection a design choice of our system, denoted by $s \in \mathcal{S} = \{S_{\text{seq}}, S_{\text{roll}}, S_{\text{par}}\}$.

Each strategy $s \in \mathcal{S}$ specifies how inference and training workloads are scheduled on the underlying hardware during the update phase, as follows:

• *Full suspension* (S_{seq}): the inference capability of all EEs is temporarily disabled, and all incoming traffic is routed directly to the main branch. The system then allocates all available resources to retrain the all affected EEs at the same time. As a consequence, inference latency increases to the full backbone latency, but, since all computation resources at the EEs can be dedicated to retraining, the retraining time is minimized. Thus, the S_{seq} strategy minimizes retraining duration but produces the highest instantaneous latency penalty.

• *Alternating Updates* (S_{roll}): EEs are updated sequentially, one at a time; during the update of exit k, only that exit is suspended. Other exits continue serving requests, reducing latency compared to full suspension. This ensures partial system availability and balances latency and retraining time.

• *Parallel execution* (S_{par}): it achieves zero downtime via *model duplication*. It creates a shadow copy of the EEs to retrain, while the original instance continues to serve inference requests using old weights. Once training is complete, the weights are atomically swapped. This minimizes service interruption but doubles the memory footprint for the target exit and induces resource contention, potentially causing latency jitter. This strategy is thus crucial for scenarios involving minor drifts or high-availability requirements.

As the first step, we need to determine the set of EEs $\bar{\mathcal{K}}{\subseteq}\mathcal{K}$ that are affected by the popularity drift to such an extent that retraining is needed. This can happen if the set $\mathcal{P}^k(t)$ of priority classes for the EE k has changed, which can be detected as explained in Sec. IV-A. Once $\bar{\mathcal{K}}$ is identified, the choice of the retraining strategy depends upon three factors: memory, cost, and inference latency. For memory, there is a hard constraint, i.e., the memory footprint $\mu^k(s)$ incurred by strategy s at EE k cannot exceed the available memory M^k:

$$\mu^k(s) \leq M^k, \quad \forall k \in \bar{\mathcal{K}}. \quad (2)$$

As a consequence of (2), the parallel execution strategy may be infeasible in some cases. Indeed, under that strategy, the required memory is given by the *sum* of the memory required by EE k in training and inference, i.e., $\mu^k(S_{\text{par}}) \approx \mu^k_{\text{train}} + \mu^k_{\text{infer}}$, with μ^k_{train} that can be multiple times larger than μ^k_{infer}.

Among the feasible strategies, we select the one to adopt by considering (i) its impact on the QoS parameters, most importantly, the inference latency, and (ii) the cost it incurs. Concerning the former, we indicate as λ^k_{orig} the latency associated with EE k *before* retraining, as $\lambda^k_{\text{retr}}(s)$ the one during the retraining under strategy s, as T_{retr} the duration of the update, and as α the rate at which requests arrive. Then, the QoS penalty incurred by strategy s for EE k is given by:

$$\alpha\, p_c(t) \prod_{h<k} \left(1-\pi^k_c(t)\right) T_{\text{retr}}(s) \left\{ \mathbf{util}(\lambda^k_{\text{orig}}) - \mathbf{util}\big[\lambda^k_{\text{retr}}(s)\big] \right\}, \tag{3}$$

where $\mathbf{util}$ denotes the utility of achieving a certain inference latency. In (3), the first four terms correspond to the number of inference requests affected by the retrain, and the last to the penalty incurred by each one.

Then we know the cost $\gamma^k_{\text{retr}}(s)$ of retraining EE k for a unit of time under strategy s; thus, we get:

$$\gamma^k_{\text{retr}}(s) T_{\text{retr}}(s). \tag{4}$$

Combining (3) and (4) via the balancing factor β and summing over all affected EEs, the best strategy $s^\star$ is given by:

$$s^\star \leftarrow \arg\min_{s\in\mathcal{S}} \sum_{k\in\bar{\mathcal{K}}} \beta \Bigg[\alpha p_c(t) \prod_{h<k} (1-\pi^k_c(t)) T_{\text{retr}}(s) \cdot \Big(\mathbf{util}(\lambda^k_{\text{orig}}) - \mathbf{util}(\lambda^k_{\text{retr}}(s))\Big) \Bigg] + (1-\beta)\left[\gamma^k_{\text{retr}}(s) T_{\text{retr}}(s)\right]. \tag{5}$$

V. Experimental Results

In this section, we empirically evaluate the ADEx framework, aiming at answering three key questions:

(1) *Efficacy & Mechanism:* Can ADEx successfully recover inference efficiency (i.e., EE usage) without compromising accuracy? Does the Priority Control Vector ($\mathcal{P}^k(t)$) correctly shift to reflect new popular classes?

(2) *Strategy Trade-offs:* How do the retraining strategies differ in terms of latency penalties and service disruption time?

(3) *System Cost:* What is the breakdown of resource consumption (memory, training time), and how does ADEx compare to a joint fine-tuning cost-reference baseline ?

A. Experimental setup

Model Architecture. We use a 3-head EE architecture built upon a ResNet-18 backbone [1]. EE1 and EE2 are designed as standalone residual branches. This design allows the EE heads to be trained independently or jointly with the backbone.

Per-route compute cost (GFLOPs). To provide a hardware-agnostic evaluation of resource consumption, we report the total FLOPs for each routing path on CIFAR-10 (32×32). As shown in Table I, the computational demand increases monotonically with the exit depth, hence, higher EE utilization directly translates to reduced inference latency.

Dataset and Popularity Drift. We evaluate ADEx using a non-stationary stream derived from the CIFAR-10 dataset. To simulate realistic popularity drift in edge environments, we construct two distinct data distributions, *Phase A* and *Phase B*, by sub-sampling the test set (5,000 samples each) based on three priority levels defined by a `RatioSplitConfig`: *Popular Classes* (80% sampling): 4 target classes represent the majority of the stream (800 samples/class); *Common Classes* (50% sampling): 3 classes represent intermediate frequency (500 samples/class); *Rare Classes* (10% sampling): 3 classes represent the long tail (100 samples/class).

TABLE I: Compute cost (GFLOPs) per routing path on CIFAR-10

Inference Path	GFLOPs / Sample
Path 1 (Exit 1)	0.656
Path 2 (Exit 2)	1.003
Path 3 (Main Exit)	1.272

In our implementation, the drift is simulated by swapping the semantic categories assigned to these ratios:

- Phase A (Pre-drift): Vehicles (*airplane, automobile, ship, truck*) are Popular; Animals (*bird, deer, dog*) are Common.
- Phase B (Post-drift): The distribution shifts such that *cat, frog, horse, bird* become Popular, while most vehicles drop to the Rare category (10% sampling).

Benchmarks. We compare ADEx against:

- *Static:* The model, trained on Phase A, runs on Phase B data without adaptation.
- *Joint Fine-tuning (cost reference):* A heavyweight baseline that performs conventional *joint training* for EE models by updating not only the exit branches but also the backbone parameters. Since this update cannot reliably serve inference on the same model instance, it requires a *shadow-copy/dual-model* setup to maintain service continuity (i.e., one model serves inference while a duplicated model is fine-tuned, followed by a weight swap). While service could be suspended to avoid duplication, we enforce service continuity for consistency with our execution-strategy setting. This design incurs substantial overhead in GPU memory and update-time, and is used as a cost reference rather than a practical edge solution.
- *TENT [29] and EATA [30] (TTA baselines):* We include test-time adaptation (TTA) baselines based on Batch Normalization (BN) [31], as these are widely used drop-in methods for unlabeled post-deployment shifts. In this context, BN-based adaptation (e.g., TENT [29]) typically updates the affine parameters or tracking statistics of the BN layers to align the model with the target data distribution. To contrast ADEx against *BN-only* adaptation, we evaluate three update granularities: per-sample (μ1), mini-batch (μ16), and buffered (buffered) updates (see Sec. V-C).

Adaptation budget. Unless stated otherwise, online adaptation is run with a fixed budget equivalent to 5 training epochs on the buffered post-drift data (same budget for strategies S_{seq}, S_{roll} and S_{par}). The same budget is used for the joint fine-tuning cost-reference baseline to ensure wall-clock comparability. For fairness, all resource profiling results in Sec. V are collected with training batch size $= 64$.

Hardware Configuration. Experiments were conducted on a workstation equipped with an NVIDIA GeForce RTX 4070 Ti (16GB VRAM). While this hardware exceeds the capabilities of typical edge nodes, it allows a precise profiling

of the *relative* overheads of each strategy without hardware bottlenecks obscuring the performance trade-offs.

B. Multi-stage performance analysis

We study the system across four stages of the popularity drift cycle. Fig. 4 depicts the evolution of prediction confidence and inference latency, while Table II presents the accuracy, average latency, and exit branch utilization.

Stage 1: Pre-shift. At this stage, a substantial portion of samples is routed to the initial EEs, as indicated by the higher density of points associated with EE1/EE2 in the top-row plots. The results in Table II confirm this level of efficiency, showing that over 92% of samples exit early (EE1: 59.16%, EE2: 33.38%). Confidence values for EEs decisions are generally high, suggesting that the gating mechanism frequently deems intermediate predictions sufficiently reliable. The bottom-row inference latency scatter shows a relatively compact distribution with a mean of 2.34 ms and a low variance of 0.87 ms^2, consistent with a regime where EEs routing is often triggered (i.e., samples exit processing early on).

Stage 2: Post-shift. After the popularity shift, sample routing changes markedly: the top row exhibits a pronounced shift in sample density toward the main branch (blue), while the density at EE1/EE2 diminishes significantly. Quantitatively, EE1 usage drops sharply to 16.34%, while the samples processed by the Main Branch surges to 46.14% (see Table II). This implies that fewer samples satisfy EE confidence criteria under the new popularity distribution. The variance drops slightly to 0.59 ms^2 because the majority of samples are consistently relayed to the deeper exit, reducing the stochasticity of EEs, albeit at the cost of a higher mean inference latency (3.05 ms), reflecting the increased computational cost as the majority of samples traverse the full depth of the model.

Stage 3: Online adaptation with retraining enabled. During the adaptation stage (applied using only S_{seq} in this case for brevity), the early-exit mechanism is temporarily disabled. As a result, the top row shows samples are routed exclusively to the Main Branch (all points are Blue), with no utilization of EE1 or EE2. The latency distribution exhibits a dramatic surge, showing the highest mean (6.25 ms) and variance (1.05 ms^2) across all stages. This peak latency is attributed to two factors: (1) the mandatory execution of the full backbone for all samples (due to disabled EEs), and (2) the computational interference (e.g., GPU contention) introduced by the concurrent background retraining process.

Stage 4: Post-adaptation. Following retraining, the exit utilization partially (or substantially) shifts back toward initial EEs in the top row, indicating improved confidence of intermediate classifiers on the new popularity distribution. Table II highlights that Main Branch usage drops back to 7.56%, while EE2 becomes the dominant exit (56.98%). The latency distribution correspondingly moves downward relative to Stage 2 (and Stage 3), returning to a low-mean regime of 2.61 ms with a variance of 0.71 ms^2, similar to Stage 1, demonstrating the successful restoration of system efficiency.

To keep the four-stage visualization concise, we use S_{seq} strategy as a representative example in Fig. 4 and Table II. We then compare execution strategies (strategies S_{seq}, S_{roll} and S_{par}) using aggregated post-adaptation results in Table III, and quantify Stage 3 resource/latency trade-offs in Table IV.

TABLE II: Performance in terms of overall accuracy, average and variance of inference latency, and EE utilization across different operational stages (under S_{seq} strategy)

	Acc.	**Aveg. Lat.**	Lat. Var.	**Exit Usage (%)**		
Stage	(%)	(ms)	(ms^2)	EE1	EE2	Main
Stage 1	93.78	2.34	0.87	59.16	33.38	7.46
Stage 2	91.20	3.05	0.59	16.34	37.52	46.14
Stage 3	94.04	6.25	1.05	-	-	100.00
Stage 4	93.16	2.61	0.71	35.46	56.98	7.56

C. Comparison with TTA Baselines

We next position ADEx against TTA baselines, which are often treated as drop-in mechanisms for unlabeled post-deployment shifts. However, in EE systems, TTA is not only an optimization problem but also a scheduling problem: updates can change exit routing, and concurrent training can interfere with request-serving latency.

To capture these system-level effects, Table III reports Stage 4 endpoint performance for ADEx, TENT [29], and EATA [30] under several operational modes. We also include the TENT and EATA baselines: μ1 and μ16 denote standard online BN adaptation that updates *all* BN layers in the model with micro-batch size 1 or 16. In contrast, the buffered variants are system-aligned: they freeze the backbone and update only the BN layers in the EE branches under the same 5-epoch-equivalent adaptation budget as ADEx.

Table III highlights that, under popularity drift, naive online TENT can be brittle in early-exit systems: TENT-μ1 collapses in accuracy (13.38%) despite exhibiting low latency; aggregating updates in TENT-μ16 improves stability (53.50% accuracy) but still underperforms substantially. TENT-buffered achieves a much stronger endpoint (89.50% accuracy), yet it remains below ADEx (92.52$\pm$0.86%), and its exit-usage profile stays close to the post-shift reference (main branch usage of $\approx$44%), indicating limited restoration of EE effectiveness. EATA exhibits a closely matching trend: EATA-μ1 similarly fails under per-sample updates (14.90% accuracy), and EATA-μ16 improves accuracy (82.68%) but still leaves the system in a backbone-heavy routing regime (41.14% main-branch usage). EATA-buffered reaches an endpoint comparable to TENT-buffered (89.06% accuracy; 43.88% main-branch usage), reinforcing that BN-only adaptation, even when stabilized via buffering, does not substantially re-specialize the exits to recover EE effectiveness. The performance gap between ADEx and TTA baselines (TENT/EATA) is due to the fact that BN-only updates merely align feature statistics but do not update the decision boundaries of the early classifiers to reflect the new class popularity.

D. Resource Consumption and Update Efficiency

Next, we evaluate the impact of retraining strategies on service quality. We analyze the Peak GPU Memory consumption

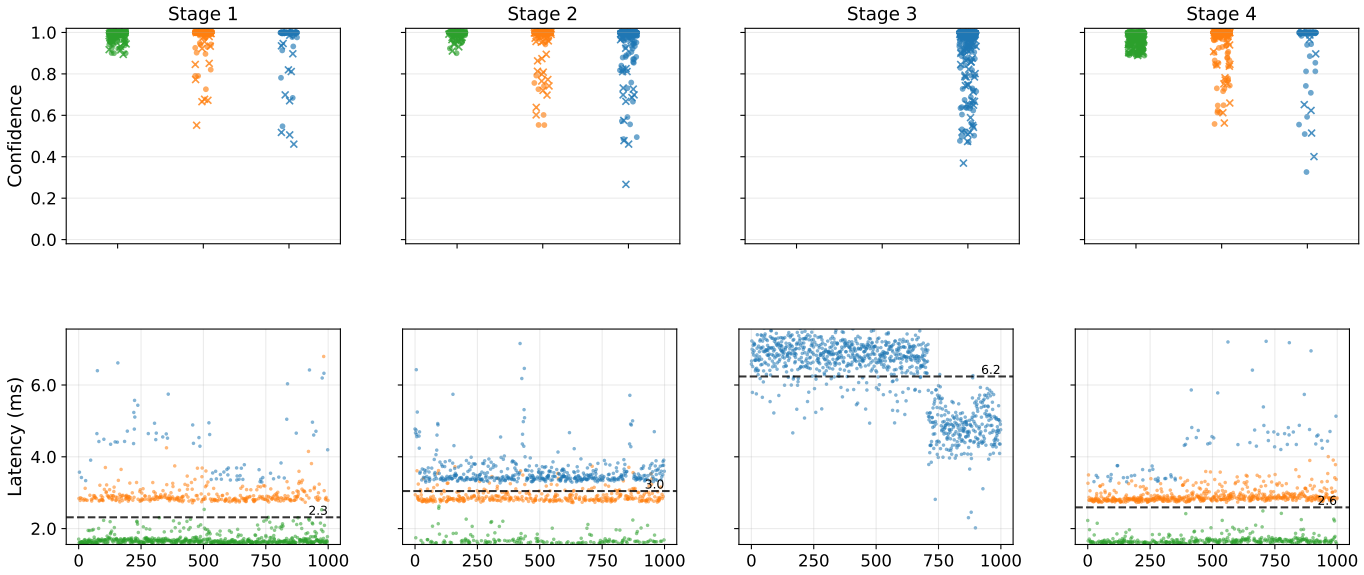

Fig. 4: Four-stage experimental results under distribution shift. Columns correspond to Stages 1–4: (1) Baseline inference on the original distribution (Phase A); (2) Inference on the drifted distribution (Phase B) before detection; (3) Active retraining using the S_{seq} strategy; (4) Inference on Phase B after adaptation. Top Row: Prediction confidence scores ($\max p$) grouped by the selected exit: EE1 (green), EE2 (orange), and Main Branch (blue). Dot markers denote correct predictions and crosses misclassifications. Bottom Row: Per-sample inference latency (downsampled to 1,000 points). The color coding follows the same exit branch mapping as the top row. Dashed horizontal lines indicate the average latency for each stage.

TABLE III: Results under popularity drift (Phase A $\rightarrow$ Phase B). Stage 1: pre-shift; Stage 2: post-shift; Stage 4 endpoint results for ADEx and TTA baselines. We report TENT/EATA under three update granularities (μ1, μ16, buffered). ADEx is aggregated across strategies S_{seq}, S_{roll} and S_{par} and reported as mean$\pm$std. Latency (mean and var.) is wall-clock per-sample system latency. Lat. Var. is computed over per-sample latencies

Stage/Approach	**Performance**			**Exit Usage (%)**		
	Acc. (%)	**Lat. (ms)**	**Lat. Var. (ms^2)**	**EE1**	**EE2**	**Main**
Ref: Stage 1 (Pre-shift)	93.78	2.34	0.87	59.16	33.38	7.46
Ref: Stage 2 (Post-shift)	91.20	3.05	0.56	16.34	37.52	46.14
ADEx (Stage 4; S_{seq}, S_{roll} and S_{par})	92.52 $\pm$ 0.86	2.57 $\pm$ 0.05	0.77 $\pm$ 0.07	38.57 $\pm$ 2.74	53.78 $\pm$ 2.79	7.65 $\pm$ 0.18
TENT-μ1 (Stage 4)	13.38	2.51	0.55	51.50	48.50	0.00
TENT-μ16 (Stage 4)	53.50	2.72	0.79	42.84	39.52	17.64
TENT-buffered(Stage 4)	89.50	2.96	0.55	18.12	37.66	44.22
EATA-μ1 (Stage 4)	14.90	2.82	0.52	29.46	70.48	0.06
EATA-μ16 (Stage 4)	82.68	3.07	0.67	22.42	36.44	41.14
EATA-buffered (Stage 4)	89.06	2.96	0.59	18.80	37.32	43.88

based on the specific operational footprints observed for each strategy (see Table IV) via PyTorch CUDA memory Profiler with a batch size of 64. The reported values reflect the total memory requirement for the update process, including the necessary allocation for model parameters and training buffers (e.g., gradients and activations).

According to Table IV, among ADEx retraining strategies, *Full Suspension* proves to be the most efficient in terms of Update Time (87.43 s), as it dedicates all available computational resources to the training task without interference from inference tasks. In contrast, *Alternating Updates* incurs the longest update duration (107.24 s) due to the serialization of the training workload. However, this serialization offers a significant advantage in memory efficiency, requiring only 317 MB of peak GPU memory – 31% less than the Parallel approach – making it the optimal choice for strictly memory-constrained edge devices. Notably, the *Parallel Execution* strategy consumes around 461 MB. This remains below a full model duplication because only the target EE modules are duplicated, while the backbone parameters are shared.

The joint fine-tuning baseline achieves the lowest wall-clock update time (76.97 s), but at a substantially higher peak GPU consumption due to backbone updates and shadow-copy deployment. This mechanism incurs higher training-side activation/gradient buffers than exit-only adaptation. To preserve service continuity, it also keeps an additional model instance for serving inference while the duplicated copy is fine-tuned, which increases peak GPU memory to 665 MB. Its mean inference latency (7.07 ms) and variance (3.29 ms^2) indicate noticeable online interference, yet remain below ADEx

TABLE IV: Performance and resource trade-offs of ADEx retraining strategies and a joint fine-tuning baseline. Latency is measured on Stage 3 inference requests during the retraining window (wall-clock per-sample latency)

Approach	Update Time (s)	Peak GPU (MB)	Avg. Lat. (ms)	Lat. Var. (ms^2)
ADEx (Full Susp.)	**87.43**	391	6.25	**1.05**
ADEx (Alternating)	107.24	**317**	**5.49**	1.11
ADEx (Parallel)	87.98	461	8.01	5.36
Baseline (Joint FT)	76.965	665	7.07	3.29

(Parallel) because inference does not simultaneously compete with two specialized EE update threads, and because routing behavior differs under the baseline's joint update mode.

In summary, our analysis confirms that no single strategy is optimal for all scenarios. While *Parallel Execution* degrades inference latency in severe drift scenarios, it ensures service continuity (zero downtime) and architectural stability. *Full Suspension* provides the fastest recovery from retraining in the case of severe drifts, while *Alternating Updates* offers a balanced, low-memory alternative.

VI. Conclusions

We introduced *ADEx*, an adaptive framework for DNNs with early exits (EE) designed to maintain efficient and robust inference under popularity drift in edge environments. Unlike approaches, ADEx aligns EE specialization with class popularity, ensuring that the most frequently occurring samples are handled by the earliest exits. It integrates local popularity drift detection, unsupervised pseudo-labeling using the main network as a teacher, and lightweight, EE-specific retraining. Experimental results show that ADEx improves accuracy after drift while restoring inference efficiency. Specifically, ADEx reduces main-branch usage from a peak of 46.14% (post-drift, pre-adaptation) to 7.65% (after ADEx adaptation) and brings the mean inference latency down from 3.05 ms to 2.57 ms. Importantly, ADEx significantly outperforms Joint Fine-tuning (Joint FT), which requires a 52.3% higher peak GPU memory footprint (665 MB vs. 317 MB) and necessitates a full shadow-copy deployment to maintain service continuity during updates. TTA baselines only achieve 89.50% accuracy and fail to recover routing efficiency, resulting in 44.22% main-branch usage.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE CVPR*, 2015.

[2] Y. Kang *et al.*, "A survey on deep neural network partition over cloud, edge and end devices," *arXiv preprint arXiv:2304.10020*, 2023.

[3] S. Teerapittayanon *et al.*, "Distributed deep neural networks over the cloud, the edge and end devices," in *IEEE ICDCS*, 2017.

[4] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *ACM ASPLOS*, 2017.

[5] M. A. Maruf and A. Azim, "Optimizing dnns model partitioning for enhanced performance on edge devices," in *Canadian Conference on Artificial Intelligence*, 2023.

[6] S. Laskaridis *et al.*, "Eenet: Learning to early exit for adaptive inference," in *ICLR Workshop*, 2021.

[7] J. Gama *et al.*, "A survey on concept drift adaptation," *ACM Computing Surveys*, 2014.

[8] S. S. *et al.*, "Model retraining upon concept drift detection in network traffic big data," *Applied Sciences*, 2025.

[9] I. Korycki and L. Krawczyk-Balska, "From concept drift to model degradation: An overview on performance-aware drift detectors," *Knowledge-Based Systems*, 2022.

[10] J. Jiang, X. Wang, M. Long, and J. Wang, "Resource efficient domain adaptation," in *ACM SIGKDD KDD*, 2020.

[11] J. Yu, L. Yang, N. Xu, J. Yang, and T. S. Huang, "Slimmable neural networks," *ArXiv*, vol. abs/1812.08928, 2018.

[12] F. Errica, H. Christiansen, V. Zaverkin, M. Niepert, and F. Alesiani, "Adaptive width neural networks," 2025.

[13] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "Skipnet: Learning dynamic routing in convolutional networks," in *ECCV*, 2018.

[14] Z. Wu *et al.*, "Blockdrop: Dynamic inference paths in residual networks," in *IEEE/CVF CCVPR*, 2018.

[15] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *ICPR*, 2016.

[16] J. Ko *et al.*, "Joint offloading and resource allocation for mobile edge computing," *IEEE Transactions on Wireless Communications*, 2018.

[17] S. Laskaridis *et al.*, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *ACM MobiCom*, 2020.

[18] E. Li *et al.*, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *ACM MECOMM*, 2018.

[19] B. Sun *et al.*, "Sapar: A surrogate-assisted dnn partitioner for efficient inferences on edge tpu pipelines," *ACM Trans. on IoT*, 2024.

[20] J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with drift detection," in *SBIA*, 2004.

[21] J. Lu *et al.*, "Learning under concept drift: A review," *IEEE Trans. on Knowledge and Data Engineering*, 2019.

[22] M. A. Al-Garadi *et al.*, "A survey of ai in edge computing," *ACM Computing Surveys*, 2020.

[23] A. R. Khouas *et al.*, "Training machine learning models at the edge: A survey," *arXiv preprint arXiv:2403.02619*, 2024.

[24] Y. Fang, P.-T. Yap, W. Lin, H. Zhu, and M. Liu, "Source-free unsupervised domain adaptation: A survey," *Neural Networks*, vol. 174, 2022.

[25] H. Rahmath P *et al.*, "Early-exit deep neural network - a comprehensive survey," *ACM Comput. Surv.*, 2024.

[26] Y. Han *et al.*, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[27] M. Ayyat, T. Nadeem, and B. Krawczyk, "Classynet: Class-aware early-exit neural networks for edge devices," *IEEE Internet of Things Journal*, vol. 11, no. 9, 2024.

[28] M. Ayyat *et al.*, "Racenet: Real-time adaptive class-aware early-exit networks for edge devices," in *IEEE PerCom*, 2025.

[29] D. Wang *et al.*, "Tent: Fully test-time adaptation by entropy minimization," in *ICLR*, 2021.

[30] S. Niu, Y. Wu, Y. Zhang, Y. Chen, Z. Lin, P. Zhao, and M. Tan, "Efficient test-time adaptation without forgetting," in *NeurIPS*, 2022.

[31] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.