ClassyNet: Class-Aware Early-Exit Neural Networks for Edge Devices

Mohammed Ayyat, *Member, IEEE*, Tamer Nadeem, *Member, IEEE*, and Bartosz Krawczyk, *Member, IEEE*

Abstract—**Edge-based and IoT devices have seen phenomenal growth in recent years, driven by the surge in demand for emerging applications that leverage machine learning models, such as deep neural networks (DNNs). However, a primary drawback of DNNs is their substantial storage/memory needs and high computational overhead, making their adoption in edge devices challenging. This limitation prompted the development of early-exit models like BranchyNet, which enable decisions to be made at earlier stages by incorporating dedicated exits within the architecture's inner layers. Nonetheless, these existing early-exit models lack control over the specific class that should exit and when. The necessity for such class-aware models is evident in numerous edge applications, where particular high-priority classes must be detected earlier due to their time-sensitive nature. In this article, we introduce ClassyNet, the first early-exit architecture designed to return only selected classes at each exit. This feature facilitates faster inference times for critical classes, allowing the initial layers to operate on edge devices. This strategy conserves considerable computational time and resources on the edge without compromising accuracy. Through extensive experiments, we show the effectiveness of ClassyNet compared to other models under various scenarios.**

Index Terms—**Class-aware classification, dynamic deep neural network (DNN), early-exit models, edge computing, on-device machine learning.**

I. Introduction

WITH a forecast of 41.6 billion edge and IoT devices by 2025 [1], and a rise of 300 million smart homes by 2023 [2], many exciting edge-based real-time applications, such as remote health care [3], augmented reality [4], and video analytics [5], are expected to increase. Edge-based devices typically host on-device machine learning models such as deep neural network (DNN) models to support these applications. However, DNN models' main drawbacks lies in high computational cost and slow processing speed. Consequently, the performance of these models will be significantly influenced by the limitations of the device's resources, such as on-device memory size. As a result, developing approaches to optimally accelerate DNN model inference computations in order to realize real-time applications on edge devices with restricted resources is very desirable.

The majority of existing work in DNN acceleration focuses on model compression, using binary weight representations, or approximate decision making. All these methods still rely on the entire deep architecture, requiring each input instance to pass through every layer for the decision to be made. In biologically inspired neural networks, heuristics are being used to reduce the processing path, effectively using only a subset of the neural network.

This led to the development of early-exit models that allow for the decision to be made on earlier stages by attaching dedicated exits to the inner layers of the architecture. Models, such as BranchyNet [6], Shallow-Deep Networks, or Patience-based early exit, use a simple classification head attached to given internal layers as a potential exit. These heads are known as internal classifiers and allow for shortening the decision-making process. If a given internal classifier displays high enough confidence, a decision is being made at this exit. Otherwise, the instance is passed to further subsequent layers and the next exit to predict the label. This allows for classifying instances without passing them through every single layer, leading to significant improvements in inference speed.

However, early-exit models do not have any control over what class should be returned by each exit. Some classes should be detected earlier than others due to their temporal importance. As an example, let us take a self-driving car. Classes responsible for collisions with other vehicles or objects should be recognized as soon as possible, while other classes corresponding to driving conditions can be recognized with some latency. The need for similar importance ordering can be observed in a plethora of other domains, such as edge computing (where some classes could be detected on devices storing only the first layers of neural networks), medicine (where life-threatening cases should be recognized as soon as possible), or intrusion detection systems (where adversarial and malicious behaviors should be isolated quickly to preserve the integrity of the system). Therefore, there is a need to develop a class-based early-exit neural network with a dedicated training procedure that will allow each potential exit to specialize in recognizing a selected subset of classes.

In this article, motivated by the above observations, we design and develop *ClassyNet*, a first early-exit architecture capable of returning only selected classes at each exit. This

Manuscript received 10 October 2023; revised 15 November 2023; accepted 5 December 2023. Date of publication 19 December 2023; date of current version 25 April 2024. This work was supported in part by the U.S. National Science Foundation under Grant OAC-2212424, and in part by the Commonwealth Cyber Initiative, an investment in the advancement of cyber research and development, innovation, and workforce development. *(Corresponding author: Mohammed Ayyat.)*

Mohammed Ayyat and Tamer Nadeem are with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA 23220 USA (e-mail: ayyatma@vcu.edu; tnadeem@vcu.edu).

Bartosz Krawczyk is with the Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology, Rochester, NY 14623 USA (e-mail: bartosz.krawczyk@rit.edu).

Digital Object Identifier 10.1109/JIOT.2023.3344120

allows for significant speedups in inference time for sensitive classes while maintaining accuracy of reference early-exit methods that do not control the ordering of classes. We summarize the contributions of this article as follows.

1) We motivate and present *ClassyNet*, a dynamic class-aware classification model for edge devices with limited resources that significantly improves the inference latency time supporting real-time applications. To the best of our knowledge, this work is the first attempt at developing a dynamic class-aware DNN model.
2) We design and develop novel loss functions, *Bag-of-Classes (BoC)* and *Cost-sensitive loss matrix*, to enable class-aware training, leading to effectively controlling the assignment of specific classes to early exits of the model.
3) We implement and evaluate the effectiveness of *ClassyNet* versus other relevant classification models via extensive experimental evaluation on real-world hardware, including edge devices, such as the NVIDIA Jetson TX2 and a workstation server, using popular image benchmarks under different scenarios and configurations of edge devices.

II. Background

A. Dynamic Neural Networks

Dynamic neural networks (DNNs) represent a burgeoning area of research in deep learning, diverging from traditional static models that maintain constant computational graphs and parameters during the inference phase. Unlike static models, DNNs possess the ability to alter their structures or parameters based on specific inputs. This adaptability confers a host of advantages, including improved accuracy, computational efficiency, and adaptiveness [7]. For example, one salient feature of dynamic networks is their ability to allocate computations on-demand during testing, selectively activating specific components, such as layers or subnetworks, depending on the input. This leads to reduced computations for straightforward samples or less informative spatial/temporal input locations. Moreover, dynamic networks boast an enhanced representation power, thanks to their data-dependent architecture and parameters [8], [9].

DNNs can be categorized into three main categories [7]: 1) sample-wise dynamic networks that adaptively infer using data-dependent models for each individual sample; 2) spatial-wise dynamic networks that tailor their processing based on various positions within text, voice, or image data; and 3) temporal-wise dynamic networks designed to adaptively process sequential data, like videos and texts, along the time dimension.

In our study, we primarily concentrate on integrating class-based classification into a particular type of sample-wise dynamic network, specifically, the early-exit neural networks.

B. Early-Exit Neural Networks

Early-exit neural networks are designed to provide an "exit" option during inference once a certain level of confidence or other criteria is reached, avoiding the need to process all layers of the network. This becomes particularly beneficial for scenarios where the input sample's complexity is lower, and the network can confidently predict its classification in its earlier layers

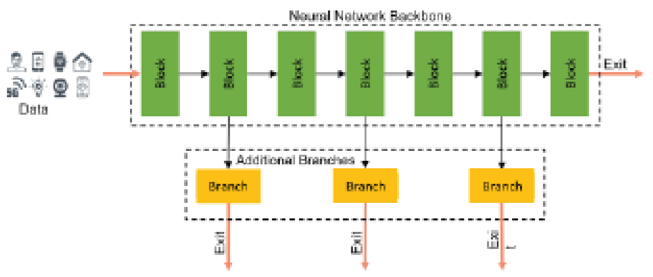

Fig. 1. Overview of BranchyNet architecture.

More formally, we consider a multiclass classification problem with K classes, where $x \in R^D$ denotes an input example and y is the corresponding target class, where $y \in S$ and $S = c_1, c_2, \ldots, c_K$, where c_i represents individual class labels in a classification problem. Let $f : R^D \rightarrow R^K$ be the neural network classifier outputting support for one of K classes. A standard early-exit model has multiple exit points E_1 $E_2, \ldots, E_M$ located on intermediate layers of f, where the last exit E_M corresponds to the traditional exit of f. Each exit E_m of the first $M - 1$ exits will be attached to the intermediate layer f_m of the base network f and will produce additional logits (also known as internal classifier supports). To produce the final output label y at any given exit, the set of logits is passed to a function to calculate the degree of confidence in the classification and compares it to a predefined threshold. If the confidence (support) value for a given instance is below that threshold at any given exit, it is assigned a label and the inference process is terminated.

C. BranchyNet Overview

BranchyNet is an early-exit solution that allows certain input samples to exit the network early by adding side branches to the original baseline network branch. This concept is based on the observation that the earlier layers of the network can correctly predict a large portion of the data population. Allowing these data points to perform an early stop and exit the network early will significantly reduce the network's overall computations, resulting in a reduction in average runtime and power consumption. We are using a variation of BranchyNet similar to [10] for IoT deployment scenarios. Fig. 1 shows the architecture overview of BranchyNet and how the side branches are added to the network backbone, enabling early inference.

The training process of BranchyNet is done by solving a joint optimization problem on the weighted sum of all the classification loss functions associated with the individual exit points. The loss function guides the learning process by evaluating how well the model performs given the input data. Each early branch is assigned a weight during training to control its relative importance. These weights are used to direct the model toward favoring specific branches.

The inference procedure begins by running the input sample through the partial network (a block or more) associated with the first exit. If the output of the exit's internal classifier is less than a given threshold, indicating high confidence, a label is assigned to the sample and the inference process is terminated. If the sample fails the exit check, it proceeds to the next exit. This iterative process is repeated until the sample exits at one of the subsequent exit points.

III. Related Works

The proliferation of DNNs and their intensive computational demands have catalyzed extensive research into optimizing and accelerating these models for edge deployments. Broadly, these strategies can be classified into static and dynamic approaches. Static methods primarily focus on shrinking the trained model's size to fit resource-limited devices. Examples encompass techniques like network compression through quantization and using smaller number of bits to represent different parameters [11], iterative pruning of nonessential DNN components such as neurons [12], substituting redundant neurons [13], and transferring the insights of a pretrained model to a more streamlined variant [14]. Local computing, another prevalent tactic, entails adjusting the DNN architecture to curtail computation without significant accuracy degradation. This includes the development of lightweight models like MobileNet [15], [16]. However, while static approaches can significantly diminish model size, they do not always guarantee a proportional boost in performance speed, especially because they often produce sparse models that cannot be easily exploited to facilitate faster results.

Dynamic inference methods [7] aim to adapt the architecture of existing neural networks during the inference process to optimize execution time, often at the expense of accuracy. These methods produce more accurate outputs when allowed extended execution time. Such dynamic approaches offer flexibility in altering the network's width, depth, or routing during runtime, making them adaptable to various use cases. Early-exiting techniques, like BranchyNet [6] and others [17], introduce exit branches after intermediate DNN layers, producing outputs akin to the final result. Other strategies include layer skipping [18], [19], where certain layers are omitted, and channel skipping [20], ignoring less significant convolutional channels.

More specifically, Recent works in early exiting have been demonstrated in DeeCap [21], which dynamically selects proper-sized decoding layers for efficient image captioning, and in the work of Bakhtiarnia et al. [22], which introduced a vision transformer architecture for early exits that increases accuracy with less overhead. Moreover, Xin et al. [23] addressed the fine-tuning strategies for early-exiting models in BERT, proposing a learning-to-exit module that extends early exiting to regression tasks. The E2CM technique [24] offers an early exit based on class means, reducing the need for extensive training and fine-tuning. Kouris et al. [25] proposed a framework for multiexit semantic segmentation networks, optimizing for efficient execution under diverse constraints. However, these papers only deal with class-agnostic dynamic models, unlike ClassyNet, which is designed to provide a class-aware early-exit model capable of dealing with some unique challenges in edge environments.

Dynamic DNN models, owing to their unique structure, are intrinsically suited for model partitioning [26]. Their multiple exit points enable individual partitions to make autonomous decisions without the need for full data processing. This methodology effectively bridges the gap between local computing and total offloading. Here, specific layers of the DNN run on the end device, sending the intermediate output to an edge server for further processing by the remaining layers. When the intermediate representation is more compact than the initial input, it can drastically curtail transmission delays relative to full offloading. A salient example is the Zero Time Waste model [27], which refines premature exit predictions by combining them with subsequent ones, ensuring expedient corrections.

Current research emphasizes the strategic positioning of self-organized exit units in early-exit models [28], as well as customizing their architecture to address specific challenges [29]. These models have gained prominence in sectors requiring swift inference, such as natural language processing [30] and video classification [31]. This highlights a difference from previous approaches by focusing on the strategic placement and organization of exit points within the network to address specific computational challenges and application requirements, rather than a one-size-fits-all approach. This nuanced method of integrating early exits suggests a trend toward more specialized and application-specific DNN optimizations, moving beyond class-agnostic models to those that are fine-tuned for particular tasks and challenges in edge computing environments.

Numerous studies have focused on the partitioning of DNN models between cloud servers and local edge devices [32]. One prominent approach is the Neurosurgeon framework [33], which optimizes the partitioning by predicting the performance of neural networks on both the local device and cloud server based on estimated processing delays and network conditions. Another approach is Edgent [34], which uses edge computing to improve DNN inference through device–edge collaboration. This framework combines model partitioning and DNN right-sizing to minimize computing latency and is adaptable to both static and dynamic network conditions, optimizing configurations based on the current bandwidth. Additionally, the multiexit DNN inference acceleration framework (MAMO) [35] focuses on multiexit DNNs to reduce latency by identifying bottlenecks in edge computing, suggesting a model that synergizes exit selection, model partitioning, and resource allocation. Experimental evaluations have shown that MAMO can significantly improve the DNN inference speed, achieving up to a 13.7$\times$ acceleration compared to contemporary methods.

Furthermore, Li et al. [36] proposed a partitioning method for the BranchyNet [6] framework. However, different from our work, they use the Branchynet framework only for choosing the DNN size. Instead of using a confidence level threshold in each side branch, their proposal uses a brute force search to choose the branch and the partition decision that achieves a given latency requirement and maximizes the inference accuracy. Pacheco and Couto [37] attempt to optimally partition a BranchyNet to minimize inference time by modeling the BranchyNet partitioning problem into a shortest path problem, which can be feasible for increasingly deeper DNNs.

In addition, combining DNN partitioning with early exit has been explored by Ebrahimi et al. [38], proposing a performance model to estimate inference latency and accuracy. Liu et al. [39] studied resource allocation for multiuser edge inference with batching and early exiting, enhancing throughput. Samikwa et al. [40] introduced an adaptive scheme for energy-efficient and low-latency machine learning over IoT networks. In the context of accelerating inference, Sun et al. [41] proposed an ensemble of internal classifiers for early exiting, improving accuracy–speed tradeoffs. However, all these works, while contributing significantly to the field, focus on accelerating class-agnostic early-exit systems, unlike ClassyNet, which is designed to provide a class-aware early-exit model capable of dealing with some unique challenges in edge environments.

The major recent development in class-aware early-exit models has been proposed by Bonato and Bouganis [42] where they proposed an augmentation scheme for pretrained models to optimally add exits to a network in order to maximize the inference of a targeted class. However, that approach is limited in two aspects. First, it designates an exit location for each class and there is no mechanism to shift classes around the exits. Second, it cannot accommodate multiple high-priority classes simultaneously. In contrast, our method offers the flexibility to select and prioritize several classes differently. Contrastingly, Duggal et al. [43] introduced an early-exiting framework for long-tailed classification, focusing on sample classification difficulty, which could complement the class-aware approach by providing a mechanism to handle classes with varying difficulty levels. However, that work is somewhat limited in scope as they mainly focus on long-tailed classification and unbalanced data sets.

IV. Motivation

In addition to the dynamic execution capabilities of early-exit-based models like BranchyNet, they tend to synergize particularly well with the concept of split deployment between the local edge device and the cloud server, providing a unique approach for edge deployment. This is due to the model's flexibility, which allows it to be designed and trained such that a portion of the model with one or more exits fits and is deployed on the edge device's memory, while the remainder of the model with the other exits is placed on the cloud server.

The problem with the current state-of-the-art early-exit-based models, including BranchyNet, is that they are class-agnostic, making them incapable of properly handling edge-specific conditions and contexts. For example, the same classification service/application may need to handle inputs from classes with different importance and sensitivity, or it may need to handle imbalanced inputs where the bulk of the input population comes from a few specific classes. The relative importance of classes is context dependant based on the user of the system and use case itself. As the same classes within the same classification problem can have different importance because of slightly different context. For example, an object classification model installed on a security camera can be trained to detect both humans and vehicles. However, based on the operation parameters, it may become more important to handle detecting humans faster or vehicles faster. This relative importance is runtime-based and is completely independent from data imbalance problems during training. Anomaly detection is another example of a classification application that handles network activity classes with varying degrees of sensitivity, such as network probing activity and an ongoing exploit for a serious vulnerability activity that has the potential for catastrophic harm. Detecting the ongoing exploit even a few milliseconds early can help the network deploy countermeasures faster and prevent large-scale network damage. The same holds true for personal health monitoring, as recognizing a heart attack is far more crucial than detecting slightly increased blood pressure. On the other hand, we can consider object recognition as an example of a classification service that may handle input from various populations in various locations. For example, although vehicles provide the majority of the input population to a parking lot camera, ships and trucks constitute the bulk of the input when this camera is deployed at a cargo loading port. Another example is a malware identification service that must be deployed at various locations, each of which suffers primarily from a distinct malware subtype.

This inspired us to develop *ClassyNet*, a framework for class-aware dynamic early-exit classification models. Class-aware models, especially when combined with model splitting, can provide several advantages, including pushing priority (important/sensitive) classes to early exits, which helps the model better achieve its operational goals by minimizing the inference path and hence the computation time. Moreover, with limited memory for edge devices, inputs from the priority classes will be processed using the partial model that is on-device and hence avoid the overhead of transmission. As our results demonstrated in Section VI-A, we can design and train models with exits placed at the very beginning of the network model that are capable of accurately classifying a substantial proportion of the targeted samples. As a result, we could build a neural network model using early-exit techniques by maintaining only a small fraction of the model on the limited edge device's memory for early inference while sending the more challenging samples to the cloud.

V. ClassyNet Architecture

A. Model Overview

The objective of ClassyNet is to develop a neural network that can cascade or stagger its outputs. Specifically, for any given exit E_m, the majority of the samples should belong to a predefined set of classes that is a subset of S. This represents a notable departure from traditional early-exit architectures, where each exit treats all classes uniformly, without any mechanism to prioritize specific classes at certain exits.

TABLE I
DESCRIPTION OF VARIABLES IN ALL EQUATIONS

Notation	Description
x	Input sample
y	Label of sample x
R^D	Domain of input dataset
R^K	Domain of possible labels
K	Number of available classes
S	Set of all classes
M	Number of exits in the neural network
E	An exit point of the neural network consisting of a portion of the network backbone and the branch
f	Output of exit E internal classifier
S_m	Set of high priority classes at exit m
C	Cost Matrix

Formally, we define multiple distinct class sets, namely, $S_1, S_2, \ldots, S_M$, each corresponding to an exit point in the network. Specifically, set S_m aligns with exit E_m. Every set is a subset of S ($S_m \subseteq S$). Notably, S_M is unique as it is associated with the network's final exit point and is, therefore, identical to S. The primary objective of ClassyNet is to ensure that for any input x with label y falling within the class set S_m, the optimal exit is at E_m. Specifically, ClassyNet strives to minimize the value of $f_m(x)$ so that it remains below the exit threshold designated for E_m.

In order to achieve this objective, we needed to fundamentally change the training process of early-exit neural network. For this purpose, we introduce two complementary additions to the training process and combination of them to enhance early-exit models.

1) *BoC:* For this approach, we aim to improve the model's ability to accurately identify the boundaries of high-priority classes at earlier exits, which should, in turn, lead to accelerated inference times for these high-priority classes. To accomplish this, we propose consolidating all nontarget classes into a single category, which we term the "class bag." All instances associated with these nontarget classes are treated as a single overarching class, and their early exit from the network is deliberately inhibited. The underlying rationale is that by simplifying the classification task at earlier exits through a reduction in the number of classes, the model should become more efficient at distinguishing high-priority classes.
2) *Cost-Sensitive Loss Matrix (Cs):* For this approach, we aim to provide an additional incentive mechanism for high-priority classes to exit earlier from the network and penalize low-priority classes. To accomplish this, we propose adjusting the classification loss during training for each sample based on the priority of its class. We achieve this by introducing a cost-sensitive loss matrix C, every pair of true and predicated labels has a designated cost. That weight is then multiplied by its raw loss value before it is reduced (averaged) across the entire batch. This allows us to penalize mistakes within certain classes as being costlier and thus emphasize the importance of correct classification of a given class at any layer of the neural network.
3) *Combined BoC and Cost-Sensitive Loss Matrix (BoC+Cs):* Here, we combine both the cost-sensitive approach with the BoC approach. In this combined approach, the class relabeling happens according to the BoC approach, and the cost matrix at each exit is built based on the new set of classes generated at that particular exit.

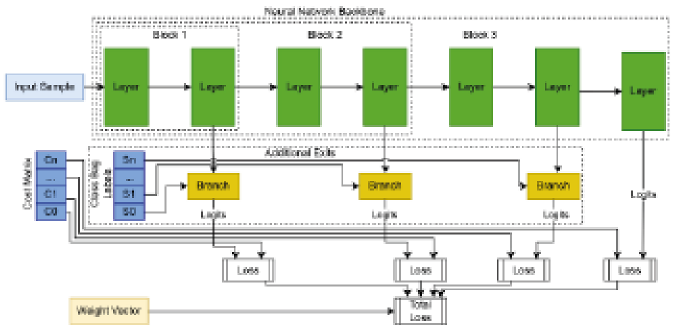

Fig. 2. ClassyNet model architecture showcasing an additional Cost matrix and sample relabeling in the additional branches.

Fig. 2 shows an overview of the ClassyNet early-exit architecture with both additional training components highlighted in light blue. The Class Bag Labels component refers to the alternative class label set assigned to each exit which maps the original class labels of the modified class labels needed for the BoC approach. The figure also highlights the addition of the cost matrix component necessary for the loss calculation at each exit. We go over the implementation details of each of these components in the following sections.

B. Bags-of-Classes Approach

1) Formal Definition: Formally, let us assume that for a given exit E_m, there is a set of high-priority classes S_m. All other lower priority classes that do not belong to S_m will be grouped as one meta-class b_m. Specifically, given an input sample $x \in R^D$ and y is its target class label, at exit E_m, the class label of x stays as y if $y \in S_m$; otherwise, the class label of x gets relabeled to b_m for this particular exit. The relabeling process of samples in shown in

$$f(y, E_m) = \begin{cases} y, & \text{if } y \in S_m \\ b_m, & \text{otherwise} \end{cases} \tag{1}$$

The relabeling of the samples using BoC should cause the f_m neural network layer with exit E_m will focus on learning the boundaries of the desired target classes in S_m and their individual boundaries against the combined boundaries of the classes within b_m. Similar to binarization approaches widely used in multiclass classification [44], this will significantly simplify the classification problem at f_m and lead to increased accuracy of labeling and correctly exiting input instances that belong to S_m.

Additionally, we identify two different schemes for constructing the subsets of classes.

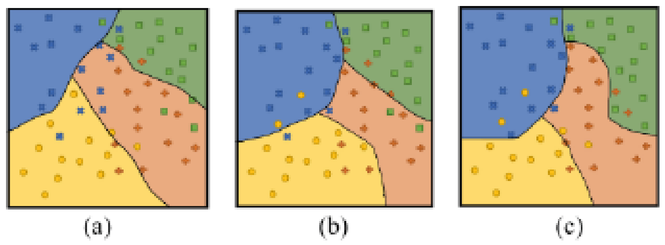
Fig. 3. Base early-exit neural network.

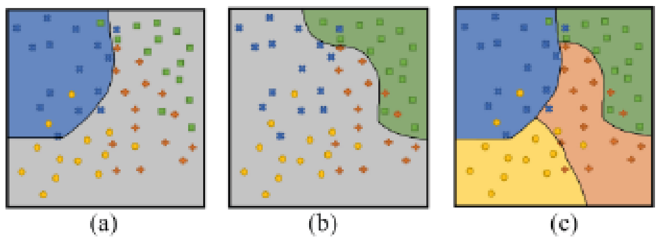
Fig. 4. ClassyNet using BoC mechanism with MC scheme. BoC is in gray shade.

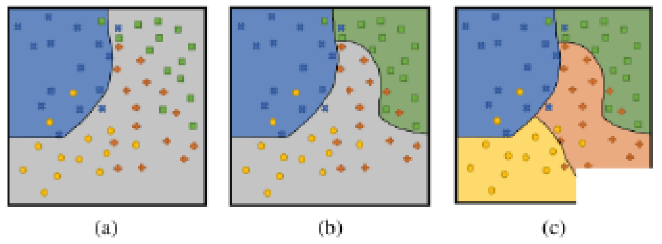
Fig. 5. ClassyNet using BoC mechanism with IC scheme. BoC is in gray shade.

1) *Mutual-Exclusive Classes (MC):* Here, class subsets at different exits are nonoverlapping, meaning that the kth class would be assigned exclusively to one subset, S_m. Formally $\forall i, j, i \neq j\ S_i \cap S_j = \phi$, and $\bigcup_{m=1..M-1} S_m \subset S$.
2) *Incremental Classes (IC):* Here, the set of classes of a subset S_m will contain include all the subsets from previous exits in addition to a new subset of classes. Formally $\forall_{m=1..M-1}\ S_m \subset S_{m+1}$.

Each approach comes with its unique advantages and drawbacks. The MC method prioritizes a specific class set for each exit, potentially sharpening the decision boundary. However, this could increase the overall inference time by restricting samples from multiple classes from exiting the network. Conversely, the IC approach often boasts quicker inference because its intermediate layers are more permissive, allowing samples from a broader range of classes to exit. However, its decision boundaries might not be as precise as those in the MC method.

2) BoC Implementation: In order to realize and build the ClassyNet with BoC mechanism, we combine and relabel as a singular meta-class all the samples of all classes that are not the target at a specific exit. This is realized by assigning additional soft labels to each training instance in the form of y_1, y_m rather than a single label y. This allows us to map the samples from the original classes into their matching classes within the class set S_m specific to each exit. During the training process, the label that matches the exit is the one that is used during loss calculation rather than the original label.

This is a flexible scheme that allows us to softly relabel the sample fairly easily while keeping the original label intact. We also modify the terminating condition so that all the samples from a given bag cannot exit the network at their nondesignated exit. The intuition behind this approach is that since the earlier exits of the network cannot recognize the differences between any of the classes in the class bag, the layer of the neural network corresponding to these exits will focus on learning the boundaries of the target classes, pushing the classes forming the bag to further, more specialized layers.

Finally, this implementation scheme provides the flexibility of being able to easily integrate or remove the BoC. As by simply assigning all classes to the high-priority set S_m at all exits, the training process reverts back to an ordinary class-agnostic training approach.

3) BoC Example: Figs. 3–5 show an illustrative example of synthetic data consisting of $K = 4$ classes and a neural network containing three exits (E_1, E_2, E_3). These sets of figures show the decision boundaries across the different exits.

The first three subplots [Fig. 3(a)–(c)] show the classification boundaries of the standard early-exit model. With Fig. 3(a) showing fairly simplistic decision boundaries across all classes. However, the decision boundaries become more sophisticated for later exits as they are deeper within the neural network. We can see that every exit deals with a rather complex multiclass decision boundary, which will significantly influence the accuracy of each early exit. Therefore, we can see a potential for the BoC approach to simplify the early-exit classification tasks.

The second three subplots [Fig. 4(a)–(c)] show the classification boundaries of the three exits in case of using mutual-exclusive class scheme (BoC-MC). In this scenario, the set of classes assigned to each exit of the network is as follows: $S_1 = \{blue\}, S_2 = \{green\}$, and $S_3 = \{blue, green, orange, yellow\}$. Fig. 4(a) shows a significantly better decision boundary [as compared to Fig. 3(a)] that focuses on discriminating only between assigned classes against all the remaining ones. This allows us to decompose the initial complex multiclass classification scheme into a much simpler problem, leading to lower model complexity and higher accuracy at this stage. Similar observations can be drawn for Fig. 4(b) representing E_2, while Fig. 4(c) is identical to Fig. 3(c), as it considers all four classes at the end.

The last three subplots [Fig. 5(a)–(c)] show the scenario when the IC scheme is used (BoC-IC) in which the set of classes assigned to each exit of the network as follows: $S_1 = \{blue\}, S_2 = \{blue, green\}$, and $S_3 = \{blue, green, orange, yellow\}$. The first exit of BoC-IC is identical to that of BoC-MC, as only a single-target class is considered. However, Fig. 5(b) shows the major difference between these two schemes, as the IC approach progressively overlaps new classes with the previous ones and builds on the boundaries of the classes from previous exits addressing previous incorrect classifications.

C. Cost-Sensitive Loss Matrix Approach

1) Formal Definition: Let us consider a random sample x, let y^k be a one-hot encoding vector of its true class label, $\hat{y}$ be the predicted output label, and C be the cost matrix of size $K_m \times K_m$, where K_m is the number of classes at any exit m. The number of classes at each exit will be different The elements of the matrix c_{ij} represent the cost associated with selecting a predicted label i when the true label is j. Considering that this is a multiclass classification problem, we decided to base our cost-sensitive loss function on the popular cross-entropy loss

$$L\left(y^k, \hat{y}\right) = -\sum_i c_{ij}\left(y_i^k \log \hat{y}_i\right). \tag{2}$$

Ordinarily, loss calculation usually requires the output logits not the label of the samples. However, to calculate the cost allocated to the sample at any given exit during the classification process, we need both the true label and the predicted label. Obtaining the predicted is fairly trivial, as we simply pick the class with the highest probability from the output logit as the predicted label.

This approach is highly inspired by [45] and other literature on instance-level and class-level costs for the loss function. Cost-sensitive learning is most commonly used to handle imbalanced data sets, where cost is seen as a penalization factor for mistakes made in minority classes. However, cost-sensitive approaches can be used beyond the imbalanced framework in order to model importance (or preferences) among classes. ClassyNet is designed to work with general multiclass problems, and thus our cost-sensitive component is used to strongly emphasize the need for highly accurate discrimination among classes expected to be returned at a given exit. This approach not only allows for an early interference of desired classes but can also potentially increase local accuracy in every individual exit. This can be combined with the BoC approach to further improve the classification of target and nontarget classes at each exit.

Another motivation for adopting the cost-sensitive loss matrix stems from addressing the limitations of the BoC approach. Namely, the BoC approach prohibits classes from exiting prior to the exit they have been assigned to. While this can help better train the model to push more of the high-priority class into earlier exits, it can adversely affect the average inference time across all classes. The cost-sensitive method offers a balanced alternative, prioritizing specific classes without entirely restricting others.

2) Cost Matrix Implementation: We implemented example-dependent cost-sensitive learning, aiming to leverage the cost information tied to class selection for each sample. This approach puts a different weight to classification errors based on the significance of the sample's class. To enact this, we introduced a distinct cost matrix at each network exit, detailing the cost for every combination of actual and predicted labels. This matrix then factored into our loss calculations. Given that classes can vary in importance at different exits, and considering the potential change in the number of classes at each exit, we found it essential to designate a unique cost matrix for every exit.

Additionally, this implementation scheme provides the flexibility of being able to easily integrate or remove the cost-sensitive matrix. As by simply assigning the cost matrix to an all-ones matrix, the training process reverts back to an ordinary class-agnostic training approach.

3) Cost-Sensitive Matrix Example: Let us consider an identical scenario as outlined in Section V-B, where we have a data set of four classes, $S = blue, green, orange, yellow$. (For simplicity, assume this is also the order of the classes.) To properly populate the values of the cost matrix C, we consider four different cases: 1) correct label for a targeted class; 2) incorrect label for a targeted class; 3) correct label for a nontargeted class; and 4) incorrect label for a nontargeted class. We assign a set of weight costs c_1, c_2, c_3, and c_4 for each case, respectively. Assuming that we want to train one of the exits to target samples from the first two classes $\{blue, green\}$, and become more efficient at classifying samples from these two classes. A cost matrix with the mutual weights of true and predicted labels of the four classes of this scenario at this exit is shown below in (3). The values of the weights are chosen to highly incentivize the training process to favor the two targeted classes. For example, the value of c_1 is always going to be lower than c_2 as we want to incentivize correct classification. The relationship between c_3 and c_4 is the same. Since we want to encourage more samples from the first two classes to exit at this exit, the values of c_1 and c_2 should also be lower than c_3 and c_4 because the former two weights represent targeted classes while the latter two are for nondesired classes

$$\begin{pmatrix} c_1 & c_2 & c_2 & c_2 \\ c_2 & c_1 & c_2 & c_2 \\ c_4 & c_4 & c_3 & c_4 \\ c_4 & c_4 & c_4 & c_3 \end{pmatrix}. \tag{3}$$

D. ClassyNet Training

The ClassyNet architecture utilizes a joint training approach that is based on a single optimization problem utilizing all intermediate exits. We achieved this by combining losses from each early-exit classifier, similar to other works in the literature. Each exit computes its own loss using cross-entropy loss, and then all the losses of the intermediate exits are weighted and summed together to compute the overall loss, which is then used for training the network.

To calculate the loss at each exit, we first need to adjust the label y of the sample x using the desired classes set S_m at exit m according to the BoC approach to produce adjusted label sample y_m that will be used in loss calculation

$$y_m = f(y, E_m) = \begin{cases} y, & \text{if } y \in S_m \\ b_m, & \text{otherwise.} \end{cases} \tag{4}$$

Additionally, we need to obtain the output probability of the sample produced from the exit $\check{y}_m$ (5), which is then normalized by passing it through the softmax function to obtain the output logits vector $\hat{y}_m$ (6)

$$\check{y}_m = f_m(x, y_m) \tag{5}$$

$$\hat{y}_m = \text{softmax}\left(\check{y}_m\right) = \frac{\exp\left(\check{y}_m\right)}{\sum_{c \in S_m} \exp\left(\check{y}_{mc}\right)}. \tag{6}$$

It is noteworthy that the length of the probability and logits vectors may vary at each exit, as the number of classes being classified can differ due to the BoC approach. The final step before calculating the loss involves obtaining the cost c associated with the sample which requires both the predicted label and the adjusted true label of the sample y_m at the exit. The predicted label y'_m is calculated by choosing the class with the highest likelihood from the logits vectors as seen in

$$y'_m = \operatorname{argmax}(\hat{y}_m). \tag{7}$$

The cost is then retrieved by getting the cost value corresponding to the true and predicted label from the cost matrix assigned to exit C_m

$$c = C_m[y_m, y'_m]. \tag{8}$$

Finally, the classification loss of the sample is calculated by calculating the cross-entropy loss of the sample and multiplying it by the value obtained from the cost matrix

$$L_m(y_m, \hat{y}_m) = -c\sum_i (y_{mi}\log\hat{y}_{mi}). \tag{9}$$

Given that, the overall loss function for ClassyNet is obtained by multiplying the loss of each exit by the weight assigned to that particular exit w

$$L_{\text{ClassyNet}}(y, \hat{y}) = \sum_{n=1}^{N} w_n L(y, \hat{y}_{exit_n}). \tag{10}$$

In these equations, we have to differentiate between two different sets of weight values: the weight assigned to an exit w which is applied to the entire loss of the exit and used to determine the importance of different exits during the training process. The weight matrix C is used to determine the relative impact of every individual sample during the training of the exits. We also implemented instance-level cost-sensitive training, where the batches of samples are trained in such a way so that if a sample from the batch were to leave from a certain exit during the training process, it would not progress into later exits, as we assign zero weight to its corresponding value in the weight matrix.

E. *ClassyNet Inference*

We can summarize the inference process of ClassyNet's, outlined in Algorithm 1, as follows. ClassyNet's classification network starts by passing the sample through the initial $Block_1$ consisting of the internal layers of the network and the branch of the first exit, producing a vector representing the classification likelihood of the sample (line 2). The vector is then normalized using *softmax* function (line 4). The cross-entropy value of the normalized probability vector is then computed at the exit point (line 5). If the entropy value is less than a predefined threshold assigned to the exit (line 6), then a label is attached to the sample and the inference process ends if either the *BoC* is not used *OR* the label belongs to the subset of classes assigned to this exit (lines 7 and 8). Note that if the model is using BoC approach and a sample is classified to belong to the BoC at an exit, it does not exit the network since it still requires additional processing to determine the particular class to which it belongs. Each of the exits is assigned a threshold using a threshold vector T prior to the inference process that defines its terminating condition. If the sample fails the exit check, it is forwarded to the next *Block* for further processing and iteratively attempts to exit at each of the subsequent exit points until the final exit in which it has to exit (line 9). The thresholds provide a means of controlling the tradeoff between the runtime and the accuracy, as exiting at higher entropy thresholds would cause more samples to exit early but would lower the overall accuracy of the model.

Algorithm 1: ClassyNet Inference Procedure

```
Input : $x$ is the input sample
Input : $M$ is the number of exits
Input : $T$ is the threshold vector
Input : $S_1, \dots, S_M$ are the subsets of classes assigned to the
        exits
1 for $m = 1..M$ do
2     $\mathcal{Z} = f_m(x)$      // outputs of layer i
3                                  // corresponding to exit i
4     $\mathcal{Y} = softmax(\mathcal{Z})$
5     $e = entropy(\mathcal{Y})$    // $entropy(\mathcal{Y}) = \sum y_c \log\hat{y}_c$
6     if $e < T_m$ then
7         if $(\neg\ BoC)$ OR $(\operatorname{argmax} y \in S_m)$ then
8             return argmax $y$
9 return argmax $y$
```

VI. Evaluation

In this section, we discuss the experiments used to evaluate the performance of ClassyNet versus other approaches under different scenarios. We split our evaluation into two main sets of experiments. The first set focuses on evaluating how successful our novel class-aware early-exit neural network (*ClassyNet*) in assigning different classes to different exits in comparison to the baseline class-agnostic early-exit network (*BranchyNet*) and other class-aware early-exit networks. The second set of experiments aims at showing the practicality and strengths of ClassyNet when utilized on edge devices in comparison to the other approaches, including BranchyNet and model compression. All the results presented in this section were obtained from training and evaluating each scenario, averaged over ten runs.

A. *ClassyNet Versus BranchyNet Experiments*

1) Experiments Design: Data Sets: In this set of experiments, we used two different data sets: 1) CIFAR-10 and 2) SVHN. The CIFAR-10 data set consists of 60 000 32×32 color images in ten classes, with 6000 images per class. The ten different classes represent airplanes, birds, cars, cats, deer, dogs, frogs, horses, ships, and trucks. There are 50 000 training images and 10 000 test images. The data set is divided into five training sets and one test set, each with 10 000 images. The test set contains exactly 1000 randomly selected images from each class. The Street View House Numbers data set, or SVHN is a digit classification benchmark data set that contains 600 000 32×32 RGB images of printed digits (from 0 to 9) cropped from pictures of house number plates. The cropped images are centered on the digit of interest, but nearby digits are kept in the image. The data set comes in two formats: 1) original images with character-level bounding boxes and 2) MNIST-like 32-by-32 images centered around a single character (many of the images do contain some noise at the sides). We used the second format in our experiments.

Classification Models: In our experiments, we evaluate the efficacy of ClassyNet by testing it on three renowned convolutional neural networks for image classification: 1) AlexNet; 2) ResNet-110; and 3) InceptionV3. AlexNet, as detailed in [46], was a pioneering approach with five convolutional and three fully connected layers. To assess early-exit architectures, we augmented it with two branches: one post the first convolutional layer of the main network and another post the second convolutional layer. ResNet-110, a variant of ResNet architecture outlined in [47], is a deep residual network with 110 layers, of which 108 are divided into three blocks. Each block houses 36 layers, with layer sizes escalating from one block to the next. We experimented with ResNet-110 using two early-exit configurations: *3_Exit*, with exits at layers #18, #72, and #110, and *7_Exit*, with exits at layers #4, #18, #36, #54, #72, #90, and #110. Finally, InceptionV3, as described in [48], is 48 layers deep but it has an intricate structure with multiple branches in its modules totalling 98 conv layers. For early-exit evaluation, we implemented a *7_Exit* configuration, positioning exits at layers #2, #5, #13, #17, #24, #36, and #48. We conducted our experiments with 3 and 7 exits for two primary reasons. First, since most contemporary methods incorporate only 1 or 2 additional branches (for a total of 2 or 3 exits), we aimed to benchmark our approach against these prevalent strategies. Second, we were keen to assess the effects of a model with a higher number of exits. Thus, we chose to evaluate our model with seven exits.

All our models were trained for 1000 epochs, with repeated runs to refine hyperparameter values. The values of our hyperparameters were obtained through this experimentation process. For all experiments, we employed the Adam optimizer at a learning rate of 0.001. For the AlexNet *3_Exit* configuration, hyperparameters are set as: weight vector [0.6, 0.2, 0.2] and exit threshold vector [0.0001, 0.005]. For the ResNet110 *7_Exit* configuration, hyperparameters are: weight vector [0.2, 0.2, 0.2, 0.1, 0.1, 0.1, 0.1] and exit threshold vector [0.3, 0.3, 0.3, 0.2, 0.2, 0.2]. Finally, for the InceptionV3 *7_Exit* configuration, they are: weight vector [0.3, 0.15, 0.15, 0.1, 0.1, 0.1, 0.1] and exit threshold vector [0.3, 0.3, 0.3, 0.2, 0.2, 0.2].

We developed two models of ClassyNet: 1) *ClassyNet_BoC* and 2) *ClassyNet_BoC+Cs*. *ClassyNet_BoC* is based on the use of the *IC* design in assigning the subset of classes to different exits as well the use of *BoC* approach in training the model. For our *3_Exit*, we assigned three classes at each of the first two exits while the last exit was assigned all other classes. As for the *7_Exit*, each of early six exits was assigned one class and the last exit was assigned the remaining four classes. *ClassyNet_BoC+C* differs from *ClassyNet_BoC* in that it employs a combination of both *BoC* and *Cost-sensitive loss matrix* approaches. For the rest of this section, we will refer to *BranchyNet* as *BN*, *ClassyNet_BoC* as *CN* and *ClassyNet_BoC+Cs* as *CN+C*. Our code implementation was built using Intel Labs distiller framework [49].

Evaluation Platform: The experiments were conducted with Python 3.6 and PyTorch 1.14. All the models used in our experiments are trained and evaluated on a workstation with 128 GB of memory, an AMD Ryzen 5950X CPU, and two Nvidia RTX 3090 GPUs.

Performance Metrics: In assessing the performance of our ClassyNet in comparison to BranchyNet, we are considering the following aspects.

1) *Exit Efficiency:* We are assessing ClassyNet's efficiency in classifying the majority of samples belonging to a set of predefined classes at a particular exit when this set of predefined classes is allocated to that exit. Because ClassyNet models use the *IC* design. We are also interested in evaluating how many of the samples who did not exit at their targeted exit are exiting at subsequent exits.
2) *Latency Time:* We are interested in understanding the impact of ClassyNet on the inference latency time. Note that the latency time in this set of experiments is solely based on the computation time on our workstation since all the models are fully deployed into the workstation memory. It is worth noting that we compute latency as a total for all of the testing data.
3) *Accuracy:* We assess the impact on classification accuracy of our novel class-aware techniques used in ClassyNet development. We achieve this by comparing the accuracy of both BranchyNet and ClassyNet models to *Baseline*.

2) Experiments Results (Exit Efficiency): Fig. 6 shows the class distributions of the different ResNet-110 models under the 7_Exit configuration and the CIFAR-10 data set. Experiments show similar figures when using the SVHN data set and the 3_Exit configuration, which we have to omit due to space limitation. The results shown in the figure are for the scenario where classes 9, 8, 6, 1, 7, and 0 are prioritized by exits 1, 2, 3, 4, 5, and 6, respectively, while the remaining classes are assigned to the last exit. This is just one of many assignment scenarios possible and our approach can handle any class assignment to any exit.

Each bar in the figure represents the class breakdown of the samples terminating at a specific exit using each method. For instance, the BranchyNet data at Exit 1 indicates that roughly 2500 samples finished processing at this stage and are spread across various classes. Of these, only 320 are from the high-priority class. In comparison, the bars for *CN* and *CN+C* display about 450 and 600 samples, respectively, and all these samples are from the prioritized class. This represents a 41% increase for *CN* and a 72% increase for *CN+C*. Looking at the *CN* distribution, we notice a sizeable improvement in the number of samples of the classes assigned to a specific exit, as the number of samples correctly classified from the assigned classes increases by up to 75% in some classes, even in the cases of classes assigned to the very early exits. This clearly shows the viability of our approach in developing class-aware models that significantly increase the number of samples of only specific classes exiting at specific exits. Finally, the

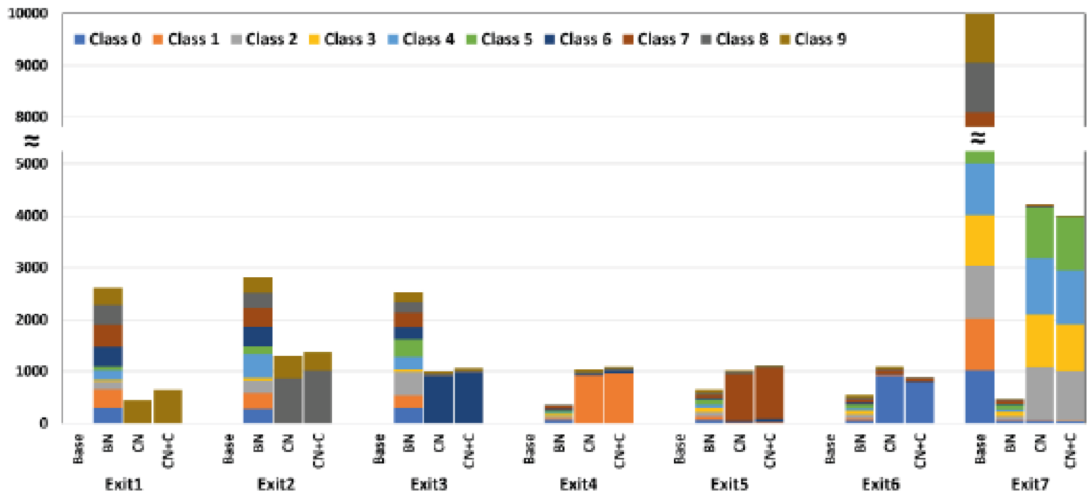

Fig. 6. Distribution of the different classes of the CIFAR-10 data set over the ResNet-110 architecture augmented with seven exists using different models.

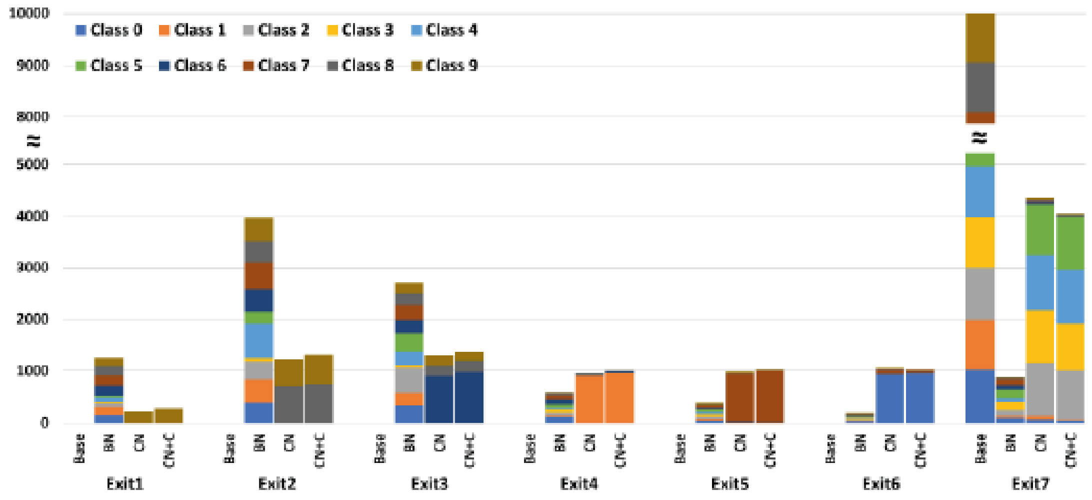

Fig. 7. Distribution of the different classes of the CIFAR-10 data set over the InceptionV3 architecture augmented with seven exists using different models.

CN+C distribution shows that adding the cost matrix to the training process almost causes the complete disappearance of unwanted classes at the subsequent exits compared to *CN*, as they are heavily incentivized to exit from their assigned exits because of the additional penalty associated with leaving from later exits.

Figs. 7 and 8 show the class distribution for the AlexNet model and InceptionV3, respectively. InceptionV3 employs the same class assignment as ResNet-110. In the AlexNet model, however, classes 2 and 3 are allocated to Exit 0, classes 4 and 5 to Exit 1, with the other classes designated to the final exit. Both figures show similar trend to the ResNet-110 model, with *CN* showing a significant increase in the number of samples from high-priority class and *CN+C* showing an even further improvement in the number of samples exiting from the high-priority class.

Finally, we asses how much faster our approach processes high-priority classes and we compare the results to the other class-specific early-exit design *(CS-EE)* introduced by Bonato and Bouganis [42]. The results are shown in Table II. Our method achieves processing speeds up to

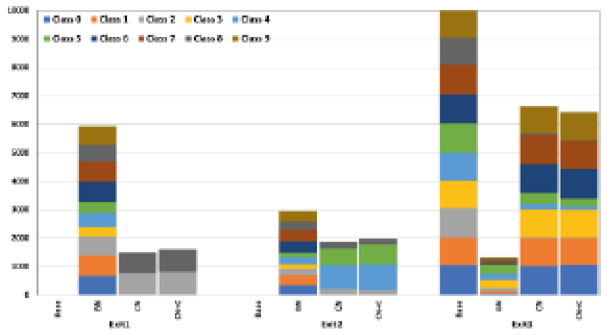

Fig. 8. Distribution of the different classes of the CIFAR-10 data set over the AlexNet architecture augmented with three exists using different models.

9.41× faster for the ResNet-110 model and 8.64× faster for the InceptionV3 model. In contrast, the *CS-EE* method only achieves speeds of 5.77× and 5.09×, respectively. Additionally, our approach maintains two distinct advantages over the *CS-EE* design. The first one is the flexibility of choosing how to assign classes to different exits as any class

TABLE II
RATE OF INFERENCE SPEEDUP OF CLASSYNET AND CS-EE COMPARED TO THE BASE RESNET-110 AND INCEPTIONV3 MODELS

	Accuracy	CS-EE	CN	CN+C
AlexNet	Maintained	-	3.45x	5.41x
Resnet-110	Maintained	5.77x	7.51x	9.41x
Inceptionv3	Maintained	5.09x	7.23x	8.64x

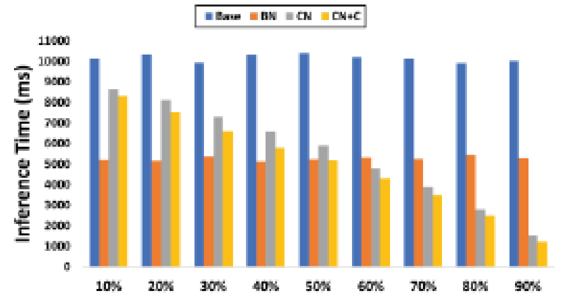

Fig. 9. Latency inference time over different compositions of the testing data.

can be prioritized at any exit, compared to the other approach where each class has an optimal exit location and there is no mechanism to shift the classes over the exit. The second one is the ability to handle several high-priority classes at once, unlike the *CS-EE* method which can only manage one class.

Latency Time: Fig. 9 shows the latency (inference) time of the different models for different compositions of testing data of a total size of 10k samples using the 7_Exit architecture and the CIFAR-10 data set. We start with a uniform distribution of CIFAR-10 testing data where each class has 1k testing samples, representing 10% of the total testing data. We manipulated the testing samples and gradually increased the percentage of the samples belonging to one target class till it reached 90% of the total testing data and measured average latency at different compositions of the testing data.

The figure shows that the latency time of both the *Baseline* and *BN* models is constant, which is to be expected given that both of these models' inference processes are independent of any manipulation of testing data. Moreover, *BN* shows a much lower latency time than the *Baseline*, which is also to be expected as the vast majority of the samples leave from earlier exits. We can also observe that *BN* outperforms *CN* in latency time when the testing data is uniform. More specifically, when the target class represents 10% of the entire 10 000 testing samples, the average latency time of *BN* is about 2500 ms (42% of *Baseline*), while for *CN*, the latency time is about 4800 ms (85% of *Baseline*). This is because *CN* limits the early exits to only specific classes and forces others to later exits, as opposed to the more open exiting criteria of *BN*. *CN+C* shows more reduction in latency time because of the cost matrix highly penalizing samples leaving from later exits.

As the composition of the data changes, with more samples belonging to the target class, the performance gap between *BN* and *CN* begins to narrow down until *CN* overtakes *BN* when nearly 50% of the samples belong to the target class. This is because *CN* is class-aware and its models are trained

TABLE III
ACCURACY RESULTS FOR THE DIFFERENT MODELS UNDER DIFFERENT ARCHITECTURES AND DATA SETS

	AlexNet (CIFAR-10)	**ResNet110 (CIFAR-10)**		**ResNet110 (SVHN)**		**InceptionV3 (CIFAR-10)**
Base	82.58%	90.21%		96.71%		92.34%
	3_Exit	**3_Exit**	**7_Exit**	**3_Exit**	**7_Exit**	**7_Exit**
BN	82.34%	90.12%	89.87%	96.56%	96.11%	92.06%
CN	82.18%	89.92%	89.63%	96.23%	95.97%	91.87%
CN_C	82.25%	89.34%	89.51%	95.71%	95.51%	91.65%

to process the targeted class to a much higher degree than *BN* with its class-agnostic approach. When the target class represents 90% of the entire testing samples, the average latency time of *CN* is about 1000 ms, which is only 40% of the corresponding latency time of *BN*.

Accuracy: Table III summarizes the classification accuracy of *Baseline*, *BN*, *CN*, and *CN+C* models for both 3_Exit and 7_Exit architectures under both data sets. From the table, we observed that both early-exit models have a decrease of less than 1% in accuracy compared to the Baseline model. However, this decrease is minimal and could be ignored. Moreover, we can observe that *CN* has a minor reduction in accuracy compared to *BN*, and this reduction slightly increases when using the more aggressive model, *CN+C*. However, all the changes in accuracy among different models are very minor as they all fall within 1% of baseline accuracy. We attribute this slight accuracy reduction to the fact that ClassyNet targets traditionally difficult samples of the target classes and attempts to adjust the model to finalize the inference process on these samples in earlier exits.

B. ClassyNet Versus BranchyNet Versus Pruning at the Edge

The objective of the experiments in this section is to evaluate the practicality and efficiency of ClassyNet when we deploy it on edge devices with limited resources. The main resources we are focusing on in our experiments are the memory requirements for model deployment and the latency of the inference computation.

1) Experiments Design: Data Sets: We are using the same data sets we used in Section VI-A, CIFAR-10 and SVHN.

Classification Models:

In addition to evaluating the edge deployment performance of *BN*, *CN*, and *CN+C* models, we also chose to compare with one of the very common approach used in edge deployment of neural networks, Network Pruning. We implement two different pruning techniques: a magnitude-based pruning (*Sensitivity Pruning*) [50] and pruning method built by exploring the High Rank of feature maps (*HRank Pruning*) [51].

Evaluation Platform: Similar to the previous experiments, we used the same tools to train and develop our models. To calculate inference measurements for edge device experiments, we used the Nvidia Jetson TX2 equipped with an ARM Cortex A-57 CPU and 8 GB of RAM. It also has a 256-core Nvidia Pascal GPU architecture with 256 NVIDIA CUDA cores. To simulate a lower end configuration, we disabled the GPU in some experiments (corresponding results are omitted due

TABLE IV
ACCURACY AND LATENCY TIME METRICS OF DIFFERENT RESNET-110 MODELS UNDER MULTIPLE EDGE DEVICE'S MEMORY SIZES SCENARIOS

	Sensitivity		HRank		BN (Uniform)		BN (Biased)		CN (Uniform)		CN (Biased)		CN+C (Uniform)		CN+C (Biased)	
Mem (Bytes)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)	Accu (%)	Time (msec)
112M	90.21	13090	90.21	13090		4651		4192		7451		1813		5827		1064
28.5M	82.13	4180	85.34	4812		15982		17891		38172		5432		31827		2452
5.54M	63.37	1525	66.16	1754	89.87	35322	88.81	38686	89.63	56723	88.37	12735	89.51	50272	88.07	4388
2.31M	25.57	968	28.71	1084		48921		49721		62823		15267		58882		7122
470K	8.13	352	15.27	378		59182		61283		75281		16084		73919		8412

to space limitations). As for model partitioning deployment, the model was split between the Nvidia Jetson environment and the workstation server environment described earlier, representing a cloud server. They were connected through a combination of wireless and wired connections, mimicking different edge–cloud network connections.

Performance Metrics: We used the *Accuracy* and *Latency Time* metrics, similar to Section VI-A. Unlike Section VI-A, the *Latency Time* in the current set of experiments refers to the overall time that involves the computation times on the edge device plus any computation time on the server, and the overhead delay associated with any data transmission needed to transfer the intermediate parameters of the model from the edge device to the server when the inference process is split between the edge device and the cloud server.

Evaluation Parameters: We evaluated the classification models under different memory sizes corresponding to different configurations of end devices. More specifically, we used 470 kB, 2.31 MB, 5.54 MB, 28.5 MB, and 112 MB memory sizes in our experiments. These memory sizes match the memory needed to deploy the intermediate set of layers of BranchyNet and ClassyNet, corresponding to exits E1, E2, E3, E5, and E7, respectively. As an example, since exit E2 is attached to the intermediate layer #18, then the partial network corresponding to E2 consists of the first 18 layers of the model. To calculate the approximate memory needed by each scenario, we start by calculating the number of parameters in the partial network. In this case, the partial network has 37 728 parameters. Since we are using 32-bit float precision variables, these parameters would need 37 728*4 = 150 912 bytes. Additionally, we are following the common practice of dividing the testing data into batches where we are using a batch size of 16, making the total memory size needed to run this model 16*150 912 = 2.31 MB. The batch size and the resultant active memory sizes were chosen to simulate different edge devices' memory restrictions. For example, 470 kB is in line with the memory requirements of TinyML [52] as well as microcontrollers with very low capabilities. On the other hand, 2.31 and 5.54 MB fit into low-end IoT devices or high-end microcotrollers, while 13.1 MB is for a mid-end IoT devices, and then 54.5 MB is for a high-end IoT/edge devices.

Since the objective of a pruning algorithm is to compress the original model to fit in the memory of the edge device, we applied both *Sensitive Pruning* and *HRank Pruning* pruning techniques to compress the ResNet-110 *Baseline* model to fit each of the different memory sizes we chose for our experiments. Using the pruning algorithms, the output models, termed as *Sensitive* and *HRank*, have 99.59%, 97.94%, 95.05%, 74.55%, and 0.00% reductions of the original Baseline model in order to fit into memory sizes of 470 kB, 2.31 MB, 5.54 MB, 28.5 MB, and 112 MB, respectively. Note that the memory size of 112 MB is large enough to accommodate the original *Baseline* model, and, therefore, no reduction is needed.

2) Experiments Results: Table IV shows the classification accuracy and the overall inference latency time for different ResNet-110 models trained on the CIFAR-10 data set. The first two columns show the performance of *Sensitive* and *HRank* models, respectively. The third column shows the performance of *BN* model under uniform testing data consisting of 1000 samples of each class, for a total of 10 000 samples. The fourth column shows also the performance of *BN* but under a biased testing set where 90% of the samples belong to the class designated to exit at the first exit of the *ClassyNet* models. The fifth and seventh columns show the performance of *CN* and *CN+C* models using uniform testing data, respectively. Similarly, the sixth and eighth columns show the performance of *CN* and *CN+C* models but with biased testing data, respectively. Note that the BranchyNet and ClassyNet models in the table are using the *7_Exit* architecture. We experimented with the *3_Exit* architecture as well as the SVHN data set. However, due to the space limitations and the similarity of the results, we are omitting the results of these experiments.

From the table, we can observe that while the latency time for both *Sensitivity* and *HRank* models decreases with lower memory size due to the reduction in the corresponding models, the accuracy declines dramatically significantly to the point where these models become unusable with very limited memory size. We can observe that the latency time of *BN* increases with lower memory sizes for both the uniform and biased testing data. More specifically, when *BN* model is deployed on the device completely (112M scenario), the inference latency time is relatively low (around 4600 ms) compared to the pruning inference time because a high percentage of samples terminate and exit at earlier exits. On the other hand, looking at the scenario when the majority of the model is deployed on the cloud (470-kB scenario), the inference time becomes extremely long due to the connection overhead of the frequent need to transfer information to the cloud for additional processing. The inference time measured

TABLE V
LATENCY TIME MEASUREMENTS (IN MS) OF DIFFERENT INCEPTIONV3 MODELS UNDER MULTIPLE EDGE DEVICE'S MEMORY SIZES SCENARIOS [*(U)* AND *(B)* REFER TO UNIFORM AND BIASED SCENARIOS, RESPECTIVELY]

Mem	BN (U)	BN (B)	CN (U)	CN (B)	CN_C (U)	CN_C (B)
1462M	6312	6447	10553	3111	9813	2088
495M	22128	23126	31688	10920	28422	8339
28.1M	62053	57748	78504	21454	74994	17153
6.52M	75366	77806	84362	29920	79857	19445
1.2M	87884	86912	111222	37812	103874	21832

includes computation time on the device, communication time between the device and the cloud, and the computation on the cloud. The communication time is high because each batch of the data might require communication with the cloud, and the average communication time per batch is in order of 200 ms. Because *BN* is class-agnostic, modifying the composition of the testing data to become skewed toward a certain target class has no effect on the inference time. Also, the memory size has no effect on accuracy because the same model is used regardless of where it is deployed.

From the table, we can observe that *CN* and *CN+C* behave similarly in the case of uniform testing data. Moreover, we can see that these models have higher latency times compared to *BN*. This is due, once again, to the fact that both *CN* models favor only a very small number of classes for early exits, forcing the rest of the classes to exit later, resulting in significant latency time due to the increased computation time as well as the overhead associated with the number of transmissions, which increases with low memory sizes. Contrary to uniform testing data, *CN* models outperform *BN* by up to 4×. For example, in the case when the model is fully deployed locally on the edge device (112-MB scenario), *ClassyNet* exhibits lower inference latency time as it is capable of exploiting the biased nature of the testing data in exiting from very early exits. On the other hand, when most of the model is deployed on the cloud (470-kB scenario), *ClassyNet* massively outperforms due to the reduced communication cost since the majority of the testing data exits from the first exit that is deployed on the edge device.

Table V shows the overall latency time for different InceptionV3 models trained on the CIFAR-10 data set. The results show similar trends to the ResNet-110 model results. For the sake of space, we opted to omit the results of both pruning algorithms and accuracy measurements.

VII. CONCLUSION AND FUTURE WORK

In this article, we designed and developed *ClassyNet*, the first dynamic class-aware classification model for edge devices with limited resources that significantly reduces inference latency time in supporting real-time applications. To the best of our knowledge, this work is the first attempt at developing a dynamic class-aware DNN model. We detailed the architecture and design details of the proposed ClassyNet, which included two novel loss functions: 1) *BoC* and 2) *cost-sensitive loss matrix* to enable class-aware training. Using real-world hardware, we compared several performance metrics of ClassyNet versus BranchyNet for different sets of testing data, exit numbers, and data sets. Furthermore, we compared ClassyNet's performance on edge devices with varied memory capacity limits to that of BranchyNet and two network pruning strategies. According to the results, ClassyNet could achieve up to 4× quicker inference latency time than the nearest model of comparable techniques.

We are currently exploring several promising avenues for future work. One key area of interest is the relationship between the number of exits in a network and the performance of ClassyNet. Determining the optimal number of exits in early-exit deep learning architectures remains an active and exciting field of research. Recent advancements include adaptive methods tailored to specific applications, such as exit strategies focused on energy efficiency [53], the development of exits that balance computational burden with predictive accuracy [54], and the adjustment of exit numbers based on window-based confidence metrics [55].

While our current article has concentrated on the integration of class-aware functionality into early-exit models, we aim to extend ClassyNet by incorporating these dynamic and innovative approaches to further enhance its performance. We are also keen on developing an automated mechanism for efficiently assigning classes to the various exits within ClassyNet. Another future direction involves applying ClassyNet to scenarios with imbalanced data sets, which we believe will ignite new and impactful research in class-aware classification.

REFERENCES

[1] (IDC Corp., Needham, MA, USA). *The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast*. Accessed: Jun. 25, 2020. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS45213219,

[2] (Statista Co., Hamburg, Germany). *Smart Home-Statistics & Facts*. Accessed: Jun. 25, 2020. [Online]. Available: https://www.statista.com/topics/2430/smart-homes/,

[3] M. Hamim, S. Paul, S. I. Hoque, M. N. Rahman, and I.-A. Baqee, "IoT based remote health monitoring system for patients and elderly people," in *Proc. Int. Conf. Robot., Elect. Signal Process. Techn. (ICREST)*, 2019, pp. 533–538.

[4] D. Jo and G. J. Kim, "AR enabled IoT for a smart and interactive environment: A survey and future directions," *Sensors*, vol. 19, no. 19, p. 4330, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/19/4330

[5] H. B. Pasandi and T. Nadeem, "CONVINCE: Collaborative cross-camera video analytics at the edge," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, 2020, pp. 1–5.

[6] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, 2016, pp. 2464–2469.

[7] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7436–7456, Nov. 2022.

[8] Y. Chen, X. Dai, M. Liu, D. Chen, L. Yuan, and Z. Liu, "Dynamic convolution: Attention over convolution kernels," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 11030–11039.

[9] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: Conditionally parameterized convolutions for efficient inference," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 1–12.

[10] S. K. Nukavarapu, M. Ayyat, and T. Nadeem, "iBranchy: An accelerated edge inference platform for IoT devices," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2021, pp. 392–396. [Online]. Available: https://doi.org/10.1145/3453142.3493517

[11] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," 2014, *arXiv:1412.6115*.

[12] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021.

[13] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021, *arXiv:2103.13630*.

[14] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.

[15] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[16] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.

[17] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, "Why should we add early exits to neural networks?" *Cogn. Comput.*, vol. 12, no. 5, pp. 954–966, Jun. 2020. [Online]. Available: https://doi.org/10.1007

[18] A. Banino, J. Balaguer, and C. Blundell, "PonderNet: Learning to ponder," 2021, *arXiv:2107.05407*.

[19] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "SkipNet: Learning dynamic routing in convolutional networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 409–424.

[20] X. Gao, Y. Zhao, Ł. Dudziak, R. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," 2019, *arXiv:1810.05331*.

[21] Z. Fei, X. Yan, S. Wang, and Q. Tian, "DeeCap: Dynamic early exiting for efficient image captioning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2022, pp. 12206–12216.

[22] A. Bakhtiarnia, Q. Zhang, and A. Iosifidis, "Single-layer vision transformers for more accurate early exits with less overhead," 2022, *arXiv:2105.09121*.

[23] J. Xin, R. Tang, Y. Yu, and J. Lin, "BERxiT: Early exiting for BERT with better fine-tuning and extension to regression," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput.*, 2021, pp. 91–104. [Online]. Available: https://aclanthology.org/2021.eacl-main.8

[24] A. Görmez, V. R. Dasari, and E. Koyuncu, "E2CM: Early exit via class means for efficient supervised and unsupervised learning," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2022, pp. 1–8.

[25] A. Kouris, S. I. Venieris, S. Laskaridis, and N. D. Lane, "Multi-exit semantic segmentation networks," 2022, *arXiv:2106.03527*.

[26] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Comput. Surveys*, vol. 55, no. 5, pp. 1–30, 2022.

[27] M. Wołczyk et al., "Zero time waste: Recycling predictions in early exit neural networks," in *Proc. 35th Adv. Neural Inf. Process. Syst.*, 2021, pp. 2516–2528.

[28] W. Ju, W. Bao, L. Ge, and D. Yuan, "Dynamic early exit scheduling for deep neural network inference through contextual bandits," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2021, pp. 823–832.

[29] X. Tan, H. Li, L. Wang, X. Huang, and Z. Xu, "Empowering adaptive early-exit inference with latency awareness," in *Proc. 35th AAAI Conf. Artif. Intell.*, 2021, pp. 9825–9833.

[30] T. Sun et al., "A simple hash-based early exiting approach for language understanding and generation," in *Proc. Find. Assoc. Comput. Linguist. (ACL)*, 2022, pp. 2409–2421.

[31] A. Ghodrati, B. E. Bejnordi, and A. Habibian, "FrameExit: Conditional early exiting for efficient video recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit., (CVPR)*, 2021, pp. 15603–15613.

[32] W. Su, L. Li, F. Liu, M. He, and X. Liang, "AI on the edge: A comprehensive review," *Artif. Intell. Rev.*, vol. 55, no. 8, pp. 6125–6183, 2022.

[33] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2017, pp. 615–629. [Online]. Available: https://doi.org/10.1145/3037697.3037698

[34] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.

[35] F. Dong et al., "Multi-exit DNN inference acceleration based on multi-dimensional optimization for edge intelligence," *IEEE Trans. Mobile Comput.*, vol. 22, no. 9, pp. 5389–5405, Sep. 2023.

[36] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," 2019, *arXiv:1910.05316*.

[37] R. G. Pacheco and R. S. Couto, "Inference time optimization using BranchyNet partitioning," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109

[38] M. Ebrahimi, A. Da S. Veith, M. Gabel, and E. de Lara, "Combining DNN partitioning and early exit," in *Proc. 5th Int. Workshop Edge Syst., Anal. Netw.*, 2022, pp. 25–30. [Online]. Available: https://doi.org/10.1145/3517206.3526270

[39] Z. Liu, Q. Lan, and K. Huang, "Resource allocation for multiuser edge inference with batching and early exiting (extended version)," 2022, *arXiv:2204.05223*.

[40] E. Samikwa, A. Di Maio, and T. Braun, "Adaptive early exit of computation for energy-efficient and low-latency machine learning over IoT networks," in *Proc. IEEE 19th Annu. Consum. Commun. Netw. Conf. (CCNC)*, 2022, pp. 200–206.

[41] T. Sun et al., "Early exiting with ensemble internal classifiers," 2021, *arXiv:2105.13792*.

[42] V. Bonato and C.-S. Bouganis, "Class-specific early exit design methodology for convolutional neural networks," *Appl. Soft Comput.*, vol. 107, Aug. 2021, Art. no. 107316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494621002398

[43] R. Duggal, S. Freitas, S. Dhamnani, D. H. Chau, and J. Sun, "ELF: An early-exiting framework for long-tailed classification," 2020, *arXiv:2006.11979*.

[44] J. Fürnkranz, "Class binarization," in *Encyclopedia of Machine Learning and Data Mining*, Berlin, Germany: Springer, 2017, pp. 203–204.

[45] C. Zhang, K. C. Tan, H. Li, and G. S. Hong, "A cost-sensitive deep belief network for imbalanced classification," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 1, pp. 109–122, Jan. 2019.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1–9. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[48] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015, *arXiv:1512.00567*.

[49] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for DNN compression research," 2019, *arXiv:1910.12232*.

[50] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," 2015, *arXiv:1506.02626*.

[51] M. Lin et al., "HRank: Filter pruning using high-rank feature map," 2020, *arXiv:2002.10179*.

[52] P. Warden and D. Situnayake, *TinyML: Machine Learning With TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. Sebastopol, CA, USA: O'Reilly, 2020. [Online]. Available: https://books.google.com/books?id=sB3mxQEACAAJ

[53] M. Bullo, S. Jardak, P. Carnelli, and D. Gündüz, "Sustainable edge intelligence through energy-aware early exiting," in *Proc. IEEE 33rd Int. Workshop Mach. Learn. Signal Process. (MLSP)*, 2023, pp. 1–6.

[54] W. Wenjian, X. Qian, X. Jun, and H. Zhikun, "DynamicSleepNet: A multi-exit neural network with adaptive inference time for sleep stage classification," *Front. Physiol.*, vol. 14, May 2023, Art. no. 1171467. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fphys.2023.1171467

[55] A. D. Gunter and S. J. E. Wilton, "A machine learning approach for predicting the difficulty of FPGA routing problems," in *Proc. IEEE 31st Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2023, pp. 63–74.

Mohammed Ayyat (Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from Cairo University, Giza, Egypt, in 2015 and 2017, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA, USA.

His current areas of interest in research include adversarial machine learning, generative adversarial networks, and network security.

Tamer Nadeem (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the University of Maryland at College Park, College Park, MD, USA.

He is an Associate Professor with the Department of Computer Science, Virginia Commonwealth University (VCU), Richmond, VA, USA. He is also the Assistant Director of the VCU Cybersecurity Center and the Founder of the Mobile Systems and Intelligent Communication (MuSIC) Lab, Department of Computer Science, VCU. Prior to VCU, he was a Senior Research Scientist with Siemens Corporate Research, Princeton, NJ, USA. He holds six U.S. patents and has more than 100 publications in peer-reviewed top scholarly journals and conference proceedings. His research interests cover several aspects of wireless networking and mobile computing systems, including smart wireless systems, mobile and edge computing, software-defined networks, machine learning for network systems, network security and privacy, Internet of Things and smart city systems, vehicular networks, and intelligent transportation systems.

Dr. Nadeem serves as a member of the technical and organizing committees of various ACM and IEEE conferences. He is currently serving on the Journal Editorial Board for *IET Communications* and *Sensors* (MDPI).

Bartosz Krawczyk (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from Wrocław University of Science and Technology, Wrocław, Poland, in 2012 and 2015, respectively.

He is an Assistant Professor with the Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology, Rochester, NY, USA, where he heads the Machine Learning and Computer Vision (MLVision) Lab. He has authored more than 60 journal papers and over 100 contributions to conferences. He has coauthored the book *Learning From Imbalanced Data Sets* (Springer, 2018). His current research interests include machine learning, continual and lifelong learning, data streams and concept drift, class imbalance and fairness, as well as explainable artificial intelligence.

Dr. Krawczyk was a recipient of numerous prestigious awards for his scientific achievements, such as the IEEE Richard Merwin Scholarship, the IEEE Outstanding Leadership Award, and the Amazon Machine Learning Award. He served as a guest editor for four journal special issues and a chair for 20 special session and workshops. He is a Program Committee Member for conferences, such as KDD (Senior PC Member), AAAI, IJCAI, ECML-PKDD, PAKDD, and IEEE BigData. He is an Editorial Board Member of *Applied Soft Computing* (Elsevier).