CACTUS: Dynamically Switchable Context-aware micro-Classifiers for Efficient IoT Inference

Mohammad Mehdi Rastikerdar, Jin Huang, Shiwei Fang, Hui Guan, Deepak Ganesan
University of Massachusetts Amherst, Amherst, MA 01003, USA
{mrastikerdar,jinhuang,shiweifang,huiguan,dganesan}@cs.umass.edu

ABSTRACT

While existing strategies to execute deep learning-based classification on low-power platforms assume the models are trained on all classes of interest, this paper posits that adopting context-awareness i.e. narrowing down a classification task to the current deployment context consisting of only recent inference queries can substantially enhance performance in resource-constrained environments. We propose a new paradigm, **CACTUS**, for scalable and efficient context-aware classification where a micro-classifier recognizes a small set of classes relevant to the current context and, when context change happens (e.g., a new class comes into the scene), rapidly switches to another suitable micro-classifier. **CACTUS** features several innovations, including optimizing the training cost of context-aware classifiers, enabling on-the-fly context-aware switching between classifiers, and balancing context switching costs and performance gains via simple yet effective switching policies. We show that **CACTUS** achieves significant benefits in accuracy, latency, and compute budget across a range of datasets and IoT platforms.

CCS CONCEPTS

• **Computer systems organization → Neural networks**.

KEYWORDS

edge computing, cloud computing, video analytics, deep neural networks

ACM Reference Format:
Mohammad Mehdi Rastikerdar, Jin Huang, Shiwei Fang, Hui Guan, Deepak Ganesan. 2024. CACTUS: Dynamically Switchable Context-aware micro-Classifiers for Efficient IoT Inference. In *The 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24), June 3–7, 2024, Minato-ku, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3643832.3661888

1 INTRODUCTION

There has been significant recent interest in executing deep learning models on low-power embedded systems and IoT devices (e.g. [38, 48, 62]). These devices are heavily resource-constrained in terms of compute and storage, and typically need to operate at very low energy budgets to ensure that they can run for weeks or months on battery power.

Prior approaches to squeeze deep learning models fall into three categories. The first is approaches to compress the model via pruning, quantization, and knowledge distillation. For example, model pruning [22] and model quantization [31, 45] techniques reduce the number of neurons in the models or the number of bits in the feature representation to achieve less compute. The second category is model partitioning, which executes the first few layers of a DNN on the IoT device and the remaining layers on the cloud [14, 16, 28, 30, 36, 60]. The third category is the early-exit inference which takes advantage of the fact that most data input can be classified with far less work than the entire model. Hence, early exit models attach exits to early layers to execute easy cases more efficiently [37, 59].

Context-aware inference: We argue that one dimension that has not been fully explored in prior efforts is *context-aware inference*. Context-aware inference narrows down a prediction task to the current context – a much shorter time window than the overall time span of the deployed task – to improve inference efficiency. Rather than attempting to squeeze the original deep learning model into memory, we instead load a smaller model that is relevant to the current context. We refer to the smaller model as a "context-aware classifier", noted as *μClassifier*.

In this work, we define "context" as a time window where only a subset of classes (called *active classes*) are likely to appear. A context change indicates a transition to a time window whose active classes are different because a new class comes into the scene. We note that other definitions of contexts are possible, but not the focus of this paper. For example, a context could also refer to a particular difficulty level of inputs for all classes such as different weather patterns (rain, fog, snow) or light conditions (bright sunlight/overexposure). These definitions are orthogonal to context changes caused by variations in active classes which is our focus.

In IoT settings, context awareness is useful in two scenarios: a) when the camera is stationary but the scene changes (e.g., different animals appear at different times of day before a wildlife monitoring camera), and b) when the camera itself moves (e.g., drones, robots, ego-centric cameras), for example, a mobile home robot could switch context when moving from one room to another. In this work, we focus on the former scenario given the rapidly growing deployment of static cameras including wildlife trailcams, backyard birdcams, and doorbell cameras. Context-aware inference is particularly effective for such static camera settings because the context tends to remain fairly static for extended periods. For example, a trail camera positioned in a forest may primarily observe the same set of animals. Similarly, a doorbell camera in a residential area is likely to capture the same types of vehicles and individuals. Context-aware inference could be beneficial in various other applications such as industrial monitoring systems where machinery

MOBISYS '24, June 3–7, 2024, Minato-ku, Tokyo, Japan

ACM ISBN 979-8-4007-0581-6/24/06.
https://doi.org/10.1145/3643832.3661888

states change infrequently or in smart agriculture where monitoring conditions may remain constant over long growing seasons. This idea holds even greater significance in environments where compute and memory resources are scarce, such as in many IoT devices. Even if the application does not require fast or real-time inference, a decreased inference time, achieved through a more efficient pipeline, proportionally reduces energy consumption on IoT devices. This is crucial because a majority of IoT devices operate on battery power, making energy efficiency vital for prolonging their operation.

Challenges: Context-aware inference presents several new research challenges. During training, we need to efficiently create a μClassifier for every possible context, but the number of contexts grows exponentially with the number of classes. For example, if there are 100 classes and we consider 3-class μclassifiers, there are C^3_{100} = 161700 μClassifers. In addition, there are many model configurations for each μClassifer, such as the input size (resolution) and the number of filters in the convolutional layers. The optimal configuration of a μClassifer depends on the similarity of classes in the context. A context whose classes have similar patterns would be harder to recognize and need to use a μClassifier that has a higher model capacity. If we consider 10 configurations per μClassifier, then nearly 1.6M models need to be trained to identify the optimal μClassifers for all possible contexts. Training μClassifiers with every configuration for each context cannot scale with a large number of configurations and classes.

At inference time, two additional challenges need to be tackled. First, we need a robust and lightweight method to detect context change so that we can switch to an appropriate μClassifier for the new context. Since context change detection lies on the critical path of every prediction, the detector needs to be very fast to avoid adding significant computation overhead to the μClassifier.

Second, after a new class is detected, we need to determine which μClassifier to switch to, i.e. what are the active classes in the new context. One option is to switch to a μClassifier that handles more classes, including the new class detected, to reduce the frequency of future context switches. However, a μClassifier with more classes tends to be more computation-intensive and thus has higher inference costs per input frame. Another option is to use a μClassifier that recognizes only the new class. This can be more efficient but will cause more frequent context switches and thus higher switch costs. Thus, we need to design a context switching policy that considers both the number of active classes in the new context and the overhead due to context switches.

Existing approaches to explore context awareness do not address all dimensions of the problem. The most relevant work [18, 21] leverages class skewness in video processing to improve efficiency, but rely on cloud servers to switch between models. This makes it less applicable to devices with limited network connectivity. Other approaches assume known class skews [57] or tailor DNN inference to specific applications without studying how to handle context changes effectively [32, 34, 42].

The CACTUS approach: We propose a novel paradigm for scalable and efficient context-aware classification that we refer to as **CACTUS**[1].

[1]CACTUS: Context-Aware Classification Through Ultrafast Switching.

Our work has three major contributions. First, we develop an *inter-class similarity* metric to estimate the difficulty in classifying the active classes in each context. We show that this metric is highly correlated with the optimal choice of configurations for the μClassifier, and hence allows us to rapidly and cheaply estimate the best configuration for each μClassifier without incurring training costs.

Second, we design an efficient context change adaptation pipeline by splitting the process into two parts – a lightweight context change detector that executes on each image as a separate head attached to each μClassifer, and a more heavyweight context predictor that executes a regular all-class classifier to identify the new class. This separation allows us to invoke the heavyweight classifier sparingly thereby reducing the computational cost.

Third, we develop a simple yet effective context switch policy that adaptively determines what size μClassifer to switch to depending on how fast active classes change in a particular deployment environment. This allows us to achieve a balance between context switching costs and the performance gain of context-aware inference. Furthermore, our system can deal with variable IoT-Cloud connectivity to enable unattended operation where the device switches between popular μClassifers that are locally stored, as well as cloud-assisted operation where the device can request new μClassifers to be created and downloaded for new contexts.

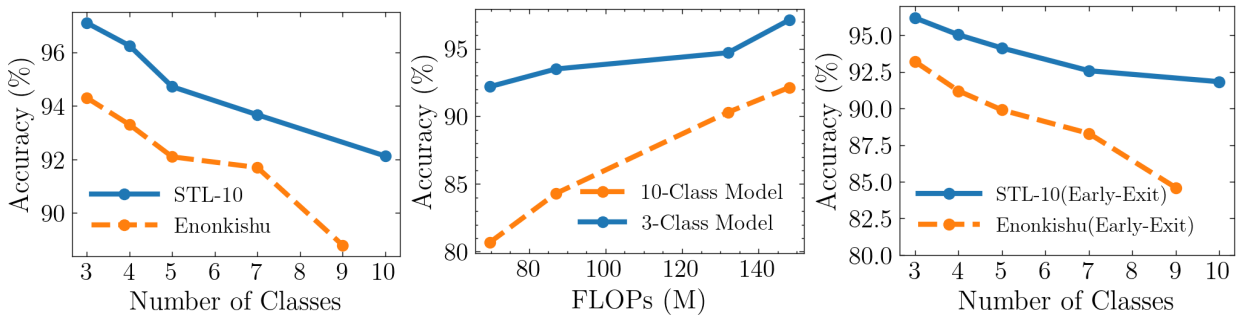
We implement **CACTUS** by modifying EfficientNet-B0 [58] and conduct extensive evaluation across five datasets (STL-10 [15], PLACES365 [64], and three Camera Trap datasets, Enonkishu [6], Camdeboo [5], and Serengeti (season 4) [7]) and two IoT processors (Risc-V(GAP8) [61] and low-end ARM 11 (Raspberry Pi Zero W) [2]). We show that:

- **CACTUS** reduces computational cost by up to 13× while outperforming All-Class EfficientNet-B0 in accuracy.
- Our inter-class similarity metric accurately captures the hardness of a set of classes in a μClassifier (0.86-0.97 Pearson's coefficient with classification accuracy).
- **CACTUS**'s context change detector has 10-20% fewer false positives and false negatives than an alternate lightweight detector that uses similar FLOPs.
- Our context switching policy leads to higher speedups in comparison to alternative policies (up to 2.5×) while the accuracy is on a par with them.
- End-to-end results on the Raspberry Pi Zero W and GAP8 IoT processors show that **CACTUS** can achieve 1.6× – 5.0× speed-up and up to 5.4% gain in accuracy compared to All-Class EfficientNet-B0.
- We also show that **CACTUS** in Local-Only setting has up to 4.1× speed-up compared to All-Class EfficientNet-B0 by storing at most 30 μClassifiers.

2 CASE FOR μCLASSIFIERS

In this section, we compare two designs for squeezing DNNs on IoT devices — the canonical method of training and compressing an *all-class model* and our approach of using a *μClassifier* that has fewer classes.

Observation 1: Using fewer classes has significant performance benefits. Intuitively, if a classifier has fewer classes than



(a) Fewer classes is better **(b) Pruning does not reduce gap (STL-10)** **(c) Early exit does not reduce gap**

Figure 1: μClassifiers provide substantial performance gains. (a) shows that the accuracy increases substantially as #classes decreases; (b) shows that model compression expands this gap further and (c) shows a similar trend for early-exit models.

another, it should be more accurate given the same resources. But how much is this gain in accuracy?

To answer this, we look at the accuracy of EfficientNet-B0 on different subsets of classes from STL-10 [15] and a Camera Trap dataset called Enonkishu [6] (See §6.1 for details on the datasets). Figure 1a shows the average accuracy across all 3-class subsets of the dataset, 4-class subsets, and so on. Post-training dynamic range quantization [1] is applied to all models. We see that accuracy increases steadily as the number of classes decreases: the overall increase is by about 5% and 5.5% for STL-10 and Enonkishu respectively when we go from 10 classes to 3 classes. This indicates the substantial accuracy advantage of μClassifiers over an all-class model given the same DNN architecture.

Observation 2: Model compression does not diminish the advantage of μClassifiers. In Figure 1a, we used EfficientNet-B0 with quantization but did not apply other techniques that are available for optimizing deep learning models for embedded MCUs (e.g. weight sparsification, filter pruning, and others [11, 13, 23, 39, 41, 46, 47, 63]). We now ask whether these techniques can bridge the accuracy gap between μClassifiers and the all-class model.

Figure 1b shows the performance gap between a 10-class and 3-class model after model pruning and quantization. We apply filter pruning [43] which reduces the number of filters in each layer based on ℓ_1 norm. Each point represents a model architecture with a specific pruning level. The accuracy of each point on the dashed line that represents a 3-class model is computed by averaging the accuracy of all 3-class subsets. We see that the accuracy gap between the μClassifier and all-class classifier increases as resources reduce (5% at 148 MFLOPs to 11.5% at 70 MFLOPs). The result highlights that model optimization techniques cannot diminish the accuracy advantage of μClassifiers over the all-class model.

Observation 3: The benefits hold for early exit models. Another approach to reduce the computational requirements of executing models on IoT devices is to use early exit, where the model only executes until it is sufficiently confident about the result and can exit without having to run the remaining layers [27, 37, 59]. This is an orthogonal dimension to the idea of limiting the number of classes, so we augment μClassifiers with early exit capability. We compare the performance of a μClassifier with early-exit capability against an all-class models with early exit.

Figure 1c shows the average accuracy across all 3-class subsets, 4-class subsets, and so on till all 10 classes after applying quantization and adding two early exit branches (stages 4 and 6) to EfficientNet-B0. We see that accuracy increases as the number of classes decreases. In particular, 3-class μClassifiers have 5% and 8.6% higher accuracy compared to all-class models for STL-10 and Enonkishu datasets respectively. Thus, the performance advantages of μClassifiers remain even if we augment the model with early exits.

3 CACTUS DESIGN OVERVIEW

We build on these observations to design a context-aware inference pipeline **CACTUS**, which executes context-aware μClassifiers for a prediction task and dynamically switches between μClassifiers to handle context changes.

CACTUS has two main components. The first component, *Configuration Predictor*, identifies the suitable configuration of the μClassifier for each context, i.e., a subset of classes (called active classes), without any training costs. It represents a context based on an *inter-class similarity* metric, which estimates the difficulty in classifying the active classes in a context. The metric allows a simple k-nearest neighbor approach to rapidly predict the optimal configuration of a μClassifier without training every possible configuration.

The second component, *context-aware switching*, enables the device to seamlessly switch between μClassifiers based on context changes. It occurs in two steps. To detect context changes, we augment μClassifiers with a lightweight *Context Change Detector* head. The first step executes the *Context Change Detector* head to rapidly detect if the input frame is a new class without determining which is the new class. If context change is detected, the second step executes the *all-class model*, a heavyweight classification module, that determines which classes are present in the scene. If a new class is detected, it triggers a "model switch" to load a new μClassifier based on the context switching policy. Figure 2 illustrates switching at run-time.

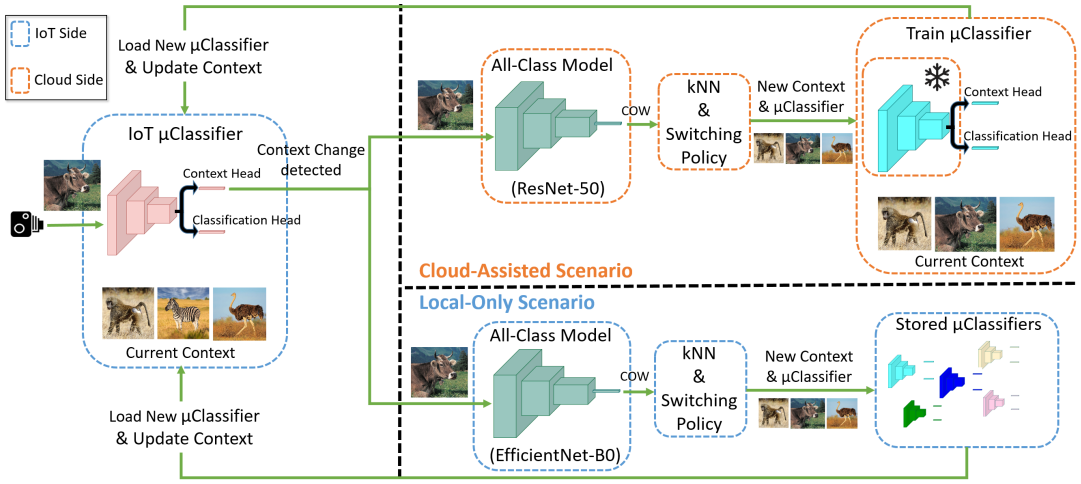


Figure 2: Context-Aware Switching in CACTUS for cloud-assisted and local-only (unattended) scenarios. In case the communication or cloud is not available, the framework relies on stored μClassifiers.

Usage scenarios: **CACTUS** supports cloud-assisted and unattended operation. When cloud assistance is available, context-aware switching detects changes in context and requests a μClassifier for the new context. **CACTUS** cloud executes the Configuration Predictor to rapidly identify the suitable configuration of the μClassifier, then trains the model on-demand and sends the model to the device to handle the new context. The new model can be cached on-device and the next time the same context occurs, the cached model can be used. For unattended operation i.e. no cloud availability, **CACTUS** relies on locally stored μClassifiers. Figure 2 shows the pipeline for both scenarios. In variable connectivity settings, a combination of cloud-assisted and unattended operation is also possible. We now describe the two components in more detail.

4 CONFIGURATION PREDICTOR

CACTUS needs to train a resource-efficient μClassifier for every possible context. As the optimal μClassifier configuration for different contexts varies, the core challenge lies in how to quickly determine the optimal configuration of the μClassifier given a context. We first give a proper definition of context in this work:

Definition 4.1 (Context). For a classification task that recognizes a set of $N = \{1, \cdots, n\}$ classes, we define a context as the time window where only a subset of classes $C \subset N$ are active (i.e., likely to appear). A m-class context means that the number of active classes is m, i.e., $|C| = m$.

To see why the optimal μClassifier configuration varies across contexts, consider classifying animals captured by a trail camera. When a context consists of animals with similar features, textures, or behaviors, the μClassifier faces a more challenging task in distinguishing between them. For instance, if the three classes represent a deer, an elk, and a moose, the classifier has to contend with the fact that all three animals have somewhat similar body shapes, sizes, and features. Consequently, the model may need to employ more computational resources to scrutinize subtle distinctions and achieve a comparable level of accuracy.

To address the challenge, our idea is to design an "inter-class similarity" metric that captures the similarity between the set of classes in a μClassifier, which in turn correlates with how much computational resources (i.e., the configuration) the model requires. Based on the metric, we design a lightweight kNN-based configuration selector to identify the best configuration of the μClassifier without needing to train all configurations. The configuration selector can be queried to output the best configuration i.e. the one that achieves the lowest resource demands while meeting a target accuracy for a set of classes (i.e., a context).

4.1 Inter-Class Similarity Metric

Given a set of classes, the inter-class similarity metric is a set of real numbers, each representing a *pairwise class similarity*. Two sets of classes that have similar statistics of inter-class similarities would share a similar μClassifier configuration. We use the statistics of inter-class similarities to represent a context, called *context representation*.

Pairwise Class Similarity: To measure the similarity between two classes, we first need to find a representation for each class. For each input image of a specific class, we get its embedding from a feature extractor that is trained on all classes. Given a set of input images of that class, we can average the embedding of all images as the *class representation*. The similarity between two classes is calculated as the Cosine Similarity between their class representations.

Context Representation: The vector representation of a set of classes (i.e., a context) is the mean and standard deviation of similarities among all class pairs. Mathematically, let $s_{i,j}$ be the cosine similarity between class i and class j. Then the representation of an m class combination C ($|C| = m$), is:

$$[\text{mean}(\{s_{i,j}\}), \text{std}(\{s_{i,j}\})], i, j \in C. \quad (1)$$

We note that the above class similarity and context representation were chosen after consideration of several alternate measures

of similarity. We show empirical results comparing these metrics in §6.3.

Figure 3 shows that the mean value of the inter-class similarity metric generally tracks the difficulty level of a context. We sample 30 different 3-class combinations from the STL-10 image classification dataset. We sort these combinations in terms of their mean similarity (Eq. 1) from low to high (X-axis). The Y-axis shows the model complexity of μClassifiers measured by FLOPs. Each entry in the heatmap reports the accuracy of the μClassifier with a specific FLOPs for a 3-class combination (darker is higher accuracy). Overall, *the higher the mean value of inter-class similarity, the less accurate a μClassifier with the same configuration (i.e., same FLOPS) since the context is more difficult.* The figure indicates that our proposed inter-class similarity metric correlates well with the computational efforts required by a context.

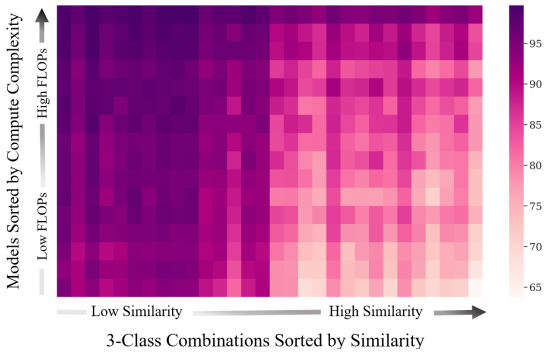


Figure 3: Model complexity versus the proposed inter-class similarity metric among classes (darker is higher accuracy).

4.2 KNN-based Configuration Predictor

The inter-class similarity metric allows us to develop a simple kNN-based approach to estimate the optimal configuration for a μClassifier. Building the kNN-based configuration predictor requires a configuration space for μClassifiers, and a set of training data points that consist of context representations and their corresponding optimal configurations.

Configuration Space: While in principle the configuration space can include any model compression method and its parameters, for practical reasons, we restrict ourselves to a few parameters to make training tractable. We therefore restrict our focus to configuration parameters that provide the large dynamic range of resource-accuracy tradeoffs for μClassifiers. We find that two methods are particularly effective for configuring μClassifiers — changing the input image resolution and the number of filters via filter pruning. In addition, we utilize post-training quantization to further lower the inference runtime and reduce the model size.

Similarity-directed sampling: The challenge of training the kNN in a resource-efficient manner is to build an effective training set. On one hand, we want samples to have good coverage of the spectrum of m-class combinations so that we can predict the best configuration of the remaining combinations accurately. On the other hand, we want as small amount of samples as possible

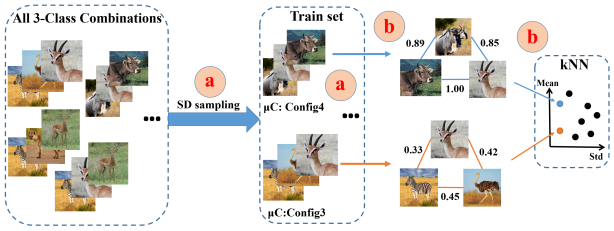


Figure 4: To build the training dataset for the kNN, we (a) use similarity-directed (SD) sampling to select x subsets of classes, and determine the best configuration for each subset as the ground truth label (e.g., Config3 and Config4); and (b) for each subset, calculate the mean and std of inter-class similarities to get its context representation.

because for each sample, we need to train all possible μclassifiers (configurations) to identify the best one.

To sample efficiently, we propose a similarity-directed sampling scheme that samples at different levels of hardness among all possible subsets of classes. Figure 4 illustrates the training of kNN. First, we compute the mean of pair-wise similarity for all combinations. We then cluster them into four groups of similarity (quartiles). We then sample randomly from each cluster to gather enough data samples for training the kNN. Compared to random sampling, our empirical evaluation on three datasets shows that our sampling scheme can reduce up to 25-66% of samples while getting similar prediction accuracy.

Querying the kNN: Once the kNN is constructed, it can be queried at run-time to predict the optimal μClassifier for the current context. The mean and std of inter-class similarities of the classes in the context are used to determine the closest configuration parameters in the kNN as the predicted optimal μClassifier configuration. Feature extractors of μClassifiers are frozen during training to save storage overhead on device and also training time on the cloud.

We note that an alternate approach to using a kNN-based configuration predictor is to use Neural Architecture Search (NAS [10, 52]). Our method is considerably lighter weight since, given a context, we can cheaply determine which configuration parameters to use via the kNN.

5 CONTEXT-AWARE SWITCHING

Another essential part of making our system work is precisely detecting context change and, once detected, rapidly switching to the right μClassifier for the new context.

This poses two challenges. The first challenge is to robustly and rapidly detect context changes. A context change occurs when an input frame falls into an unknown class that is not covered by the current μClassifier. Since context change detection lies on the critical path of every prediction, the detector should be very fast. The second challenge is to determine the appropriate μClassifier to switch to after the new class is identified. Switching to a μClassifier with less number of classes can reduce the inference cost per input frame as the μClassifier can be more lightweight. However, it can introduce more frequent context switches and thus increase runtime overhead in the future. We now explain the proposed context

change adaptation pipeline and the hybrid context switching policy that address the two challenges.

5.1 Context Change Adaptation Pipeline

We design an efficient context change adaptation pipeline by splitting the process into two parts – a lightweight *context change detector* that executes on each image as a separate head attached to each μClassifer, and a more heavyweight *context predictor* that executes a regular all-class classifier to identify the new class. This separation allows us to invoke the heavyweight classifier sparingly thereby reducing the computational cost.

Context change detector: The goal of this module is to detect when the current context has changed. The problem is essentially the same as detecting out-of-distribution samples, which falls into the area of uncertainty estimation [9, 19, 24, 40, 50, 51, 53–56]. Although many approaches have been proposed for uncertainty estimation, most of them are computationally intensive and thus cannot be applied in a real-time scenario (see §7). Among all the approaches, Maximum Softmax Probability [24] is the cheapest way to capture out-of-distribution samples. However, as we show later, this method yields unsatisfactory performance in our work due to high false positive and negative rates.

In this work, we introduce a regression-based context-change detector. Specifically, a regression head is added to the μClassifier as the second output head along with the classification head. The regressor outputs a value around 0 for the classes on which the μClassifier is trained and a value around 1 for all other classes i.e. for a new context. Both the classification and regression heads share the feature extractor of the μClassifier and thus the additional computation overhead introduced by the regressor is negligible. Since the output of the regressor is a continuous value, a threshold, θ, is needed to distinguish between the current context and the new context. This threshold is dataset-dependent and tunable considering the trade-off between the ratio of false positives and false negatives. We set $\theta = 0.5$ by default.

Context predictor: After a context change is confirmed by the context change detector, the next step is to identify the new class. This is accomplished by an *all-class* model that determines which are the classes that need to be included in the new μclassifier. The *all-class* model can be either executed locally or at the edge cloud depending on connectivity. Executing the *all-class* model at the edge cloud allows us to use more powerful and more accurate models for determining the classes in the current context, which in turn can lead to more precise switching.

5.2 Hybrid Context Switching

After a new class is identified, the challenge is to decide what size μClassifier to switch to. The tradeoff is that using μClassifiers with a larger number of classes can reduce future context switches and improve accuracy whereas using μClassifiers with a small number of classes can reduce inference latency per input frame.

Our insight are two-fold. First, the right balance depends on the deployment environment: A device that encounters much more active classes than another device is likely to prefer μClassifiers with more classes to avoid higher context switch overheads. We introduce a configuration parameter that can be easily tuned based on the target deployment environment to determine how the μClassifiers are switched.

Second, we do not need to consider all possible μClassifier sizes since the computational benefits of μClassifiers drops dramatically as the number of classes in them increases. Figure 5 illustrates this using the PLACES20 dataset, which consists of a total of 20 classes from PLACES365 [64]. For each combination of m classes, we select the most efficient configuration that meets a pre-defined accuracy threshold. Then we calculate an average of the computation savings compared to the all-class model for all possible combinations of m classes. On the X-axis, "Max" denotes the maximum computation savings, achieved by consistently using the lowest FLOPs configuration. Generally, the computation saving of μClassifiers starts to decay as we increase the number of classes they cover and at some point, they provide no computation saving. Here, as the number of classes in a μClassifier goes beyond ten, the μClassifier brings no computation savings compared to the full-class model. This motivates us to prioritize μClassifiers with a small number of classes (e.g., 2 to 5) to capitalize on the computation savings from context-aware inference.

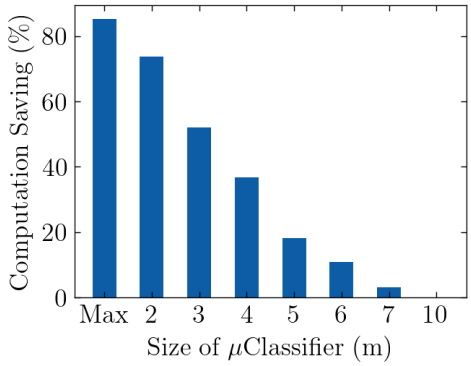


Figure 5: Computation saving vs μClassifier size (m).

Based on the two insights, we develop a "hybrid" context switch policy that switches between μClassifers based on the deployment environment.

Our policy works as follows. A kNN is already trained for each $m \in \{2, 3, 4, ...\}$ with a predetermined accuracy threshold. We start by looking at the configuration needed for the m-class μClassifier (i.e. context size = m) where $m \in \{2, 3, 4, ...\}$. We then pick the context (μClassifier) with the most efficient configuration. If multiple context sizes share the same configuration, we select the largest context size that corresponds to this configuration. This allows us to pick a minimal μClassifier configuration while also minimizing future context switches by picking as large as possible context size.

We now explain two optimizations to efficiently implement the context switching policy in both cloud-assisted and unattended operation modes.

Caching μClassifiers: In cloud-assisted mode, a device downloads the μClassifier heads for a new context and caches them on device to avoid model training for the same context. If the cache is full due to device storage constraints, the least frequently used model will be replaced. Model caching reduces the overhead of cloud-side model training and device-cloud communication.

In unattended mode, a set of μClassifiers will be pre-installed on the device before deployment. The context switching policy will switch between only the pre-installed μClassifiers. We select the μClassifiers that are most frequently used based on an input trace collected from the IoT device deployed in a target environment.

Optimizing storage requirements: μClassifiers that operate on the same model size share the same model weights as their feature extractors and are frozen during training. We can reduce storage requirements by storing one feature extractor per model size, and only storing the classification and context change detection heads for each chosen μClassifier.

Specifically, in cloud-assisted mode, a device stores all the unique sized feature extractors, one classification and context change detection head for the current context, and if storage is available, additional classification and context change detection heads from previous contexts.

In the unattended mode (i.e., without cloud availability), a device stores all the unique sized feature extractors, an *all-class* model for context predictor, and, subject to storage capacity, classification, and context change detection heads from chosen contexts based on the trace of the training data. These cached contexts are the most frequent when we apply the hybrid context-switching policy on the training set. We show in § 6.5 that the total amount of storage required for 30 μClassifiers plus feature extractors and the *all-class* model is only 23.3MB.

6 EVALUATION

We first describe the experiment settings in § 6.1, and then evaluate the end-to-end system over five datasets and on real platforms in § 6.2 and the components of **CACTUS** in § 6.3-6.5. We further perform ablation studies on context change frequency and scalability of the system in § 6.6 and implement a camera trap application in § 6.7.

6.1 Experiment Settings

Dataset: We use five datasets. Three datasets come from the Camera Trap applications called Enonkishu [6], Camdeboo [5], and Serengeti (season 4) [7]. We refer to them together as Camera Trap datasets. They have temporal data captured by trail cameras with real-world context changes. Trail cameras make use of an infrared motion detector and time-lapse modes to capture images, each time in a sequence of three. The Camera Trap datasets normally include a lot of species plus an empty class whose images show a scene with no species in it. Not all species have enough samples for both train and test sets. Hence we considered the most frequent species plus the empty class for the classification task. The number of considered classes were 9, 11, and 18 for Enonkishu, Camdeboo, and Serengeti respectively.

We also evaluate on two other image datasets, STL-10 [15], and PLACES365 [64], from which we synthesize temporal sequences. STL-10 is an image recognition dataset consisting of RGB images with a size of 96×96. The dataset contains 10 classes, each with 1300 labeled images. PLACES365 is intended for visual understanding tasks such as scene context, object recognition, and action prediction. We chose 100 scene categories, each with 1200 samples of size 256×256. While Most results are shown for 20 categories (called PLACE20), we evaluate **CACTUS**'s scalability for 100 classes.

KNN-based configuration predictor: The μClassifier is based on the feature extractor of EfficientNet-B0 [58] pre-trained on ImageNet. We vary both the pruning level and the resolution of input frames to get different configurations of the feature extractor. Specifically, EfficientNet-B0 features 9 stages, each with a predetermined output channel size. For pruning level p, we scale down the output channel size of each stage (except the first) by a factor of $1 - p$, creating a modified model. We then prune the filters with the lowest $\ell 1$ norm from each layer of EfficientNet-B0 to align with the filter count of the corresponding layer in the new model (new weights). Then the new model is initialized with this new set of weights. We applied three levels of pruning (p), 10%, 30%, and 40%, so that including the EfficientNet-B0, we have a total of four unique-sized feature extractors. The range of image resolutions is considered 100–320, 58–96, and 130–256 for Camera Trap, STL-10, and PLACES datasets respectively. Overall, the size of the configuration space is 16. A μClassifier consists of one of the four feature extractors and two heads, one for classification and one for regression (to detect context change). Each head uses two Fully Connected layers. Feature extractors are fine-tuned once on all classes and remain frozen during the training of μClassifiers' heads.

To build the training data for the configuration predictor, we applied similarity-directed sampling (detailed in §4.2) on m-class combinations. For each sampled m-class combination, we train μClassifiers with all 16 configurations to select the most efficient one that achieves a target accuracy (i.e., the optimal μClassifier configuration). We further use the feature extractor of ResNet-50 to extract the embedding of images from a class and average the embeddings into a single vector to compute context representation as Eq. 1. The pairs of context representation for the sampled m-class combinations and the optimal μClassifier configuration for the context are the training dataset of the kNN.

To balance computation efficiency and accuracy, the target accuracy threshold for selecting the best μClassifier is set to be around 94%, 87%, 92%, 92%, and 97% for Enonkishu, Camdeboo, Serengeti, STL-10, and PLACES20 datasets respectively, unless noted differently.

After collecting the training data, we use the kNN to predict the best μClassifier configuration for the remaining m-class combinations. kNN predicts the configuration of a μClassifier based on the majority vote. It starts with 3 nearest neighbors and if there is no majority, it considers 4 nearest neighbors and so on until a majority is achieved.

Context-aware switching: As discussed in §5, a regression head is attached to a μClassifier to detect context change. Once context change is detected, we execute an all-class model (ResNet-50 for the edge-cloud assisted setting and EfficientNet-B0 for unattended op.) to identify which new class occurs in the scene. All-class models are pre-trained on ImageNet and fine-tuned on the target dataset.

Implementation: We implement the **CACTUS** pipeline on two different IoT devices — Raspberry Pi Zero W, and GAP8. On both platforms, we profile the execution latency of models of different configurations on different datasets. For profiling, We generate **TensorFlow Lite** models using the default optimization and int-8

quantization to reduce the model size and execution latency. For profiling on Raspberry Zero W, we utilize the **TVM** framework [12] to tune the operator implementations to reduce the execution latency for the Arm Cores. For profiling on GAP8, we use the GAP8 tool chain [3] to transform the **TensorFlow Lite** models into C code by fusing the operations. For the end-to-end implementation on Raspberry Pi Zero W (§ 6.7), we use the ONNX quantized version of the models as it provides lower latency on the device without **TVM** tuning. We utilize Joulescope JS110 [4] to measure the energy consumption in § 6.7. A cluster of NVIDIA Tesla M40 24GB GPUs was used for training models. The source code is available at GitHub.

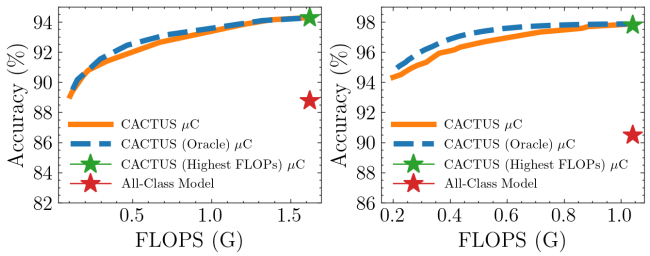
6.2 End-to-End Evaluation

This section looks at an edge-cloud assisted system (the unattended setting is examined in § 6.5). To evaluate the end-to-end performance of **CACTUS**, we need datasets that have context changes. The three Camera Trap datasets provide us with a real-world temporal flow with context changes but other datasets are not temporal. Hence, we sequence test sets from the PLACES and STL-10 datasets to mimic the temporal flow of the Camera Trap to create a temporal sequence with which to evaluate our methods. Specifically, we find that for Camera Trap datasets considering 3-class contexts, the average context interval is 30 frames (context changes in every 30 frames). We thus synthesize a test set with the context interval of 30 frames for the PLACES and STL-10 datasets and use this in our evaluation unless noted differently. In § 6.6, we further study the effects of different context change rates in the performance of **CACTUS**.

We compare the classification accuracy and overall speedup of **CACTUS** against the following baselines: (a) **FAST** [57], a context-aware approach that assumes the class skews are known offline. During inference it detects the class skew using a window-based detector and train a specialized classifier (top layers), (b) **PALLEON** [18], a state-of-the-art video processing framework that applies a Bayesian filter to adapt the model to the context. The Bayesian filter is determined by applying an all-class model on a window of frames. (c) **All-Class Model**, which uses the well-trained EfficientNet-B0 with all the classes and full image resolution. (d) **All-Class-Early Exit**, which is the early exit variant (exit heads in stages 4 and 6) of the All-Class Model. (e) **All-Class-Pruned**, where we apply 30% pruning to the all-class model. We chose 30% as it gives the best accuracy and efficiency trade-offs.

Table 1 shows the end-to-end results on two devices. The IoT device communicates at a data rate of 3 Mbps (typical LTE speed). Computed latencies include the time spent on the inference on the device, transmission of the triggered frames to the cloud, and transmission of the classification and regression heads to the device. We assumed no μClassifiers are cached on the device which is the *worst-case scenario* as it requires more transmission and incurs higher latency. We averaged latencies over the test set and computed the speedup relative to the All-Class Model. Note that for the Camera Trap datasets, accuracies are generally low regardless of whether we use EfficientNet or ResNet-50 due to data cases that are difficult to classify as well as inaccurate labels.

Overall, our approach has the advantage in terms of accuracy and latency over baselines. When compared to All-Class Model,



(a) Enonkishu **(b) PLACES20**

Figure 6: Performance of configuration predictor with different accuracy thresholds.

CACTUS has up to 5.4% higher accuracy and also offers latency speedup of 1.6 – 3.5× for the Pi0 and 2.0 – 5.0× for the GAP8. **CACTUS** outperforms two other context-agnostic baselines, All-Class-Early Exit, and All-Class-Pruned, in terms of both accuracy and latency by leveraging context awareness to improve efficiency.

Compared to the context-aware approaches FAST [57] and PALLEON [18], **CACTUS** demonstrates comparable accuracy on the Camera trap datasets, and 3.7% – 4.9% accuracy improvement for the PLACES20 dataset, but has 1.6× – 5.0× speedup. FAST and PALLEON need to execute the All-Class model for several frames to determine the class skew. Thus, they provide speedups only if the context change is very infrequent. For this reason, their latency speedup is at most 1.3× which is significantly lower than **CACTUS**. Although Table 1 reports no accuracy gains for **CACTUS** on the Camdeboo and Serengeti datasets, it is possible to achieve higher accuracy by adjusting the trade-off between accuracy and inference speed: a higher accuracy can be obtained by increasing the accuracy threshold when training the kNN model, which in turn lowers the inference speedup as more heavyweight μClassifiers will be selected.

We also profiled the peak memory consumption for different configurations using TensorFlow Lite benchmark tools [8] on a Raspberry Pi0. For camera trap datasets, μClassifiers showed a memory usage range of 11.5-43.9MB, whereas the All-Class model required 41.3MB. For the PLACES dataset, μClassifiers' peak memory consumption varied between 13.5-31.8MB, while the All-Class model demanded 30MB.

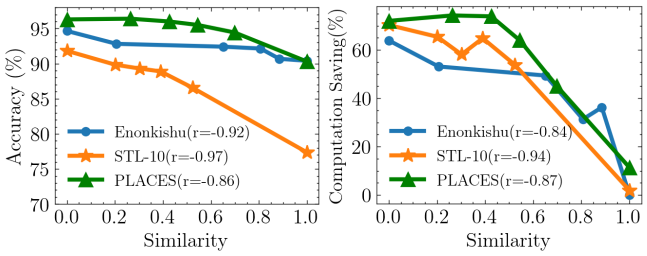
6.3 Performance of Config Predictor

We now look at the performance of the μClassifier *Configuration Predictor*. We vary the target accuracy threshold from the lowest to the highest of all the trained μClassifiers to show how the threshold affects the performance of the configuration predictor. A lower target accuracy will result in lighter μClassifiers and thus less FLOPs.

Figure 6 shows how different target accuracy thresholds affect the classification accuracy and the computation cost for our method and baselines. We compare the **CACTUS**'s configuration predictor (noted as **CACTUS μC** in the figure) with three baselines: (a) **CACTUS (Oracle) μC** i.e. the μClassifier that meets the target accuracy threshold with minimum FLOPs (which would be selected by an omniscient scheme rather than the kNN-based method we use). We note that training and storing the Oracle is not scalable — For

Methods	Enonkishu			Camdeboo			Serengeti			PLACES20		
	Acc	Pi0	GAP8	Acc	Pi0	GAP8	Acc	Pi0	GAP8	Acc	Pi0	GAP8
All-Class Model	88.9%	1.0×	1.0×	65.4%	1.0×	1.0×	74.3%	1.0×	1.0×	90.5%	1.0×	1.0×
All-Class-Early Exit	84.6%	1.7×	2.0×	65.5%	1.1×	1.2×	68.7%	1.5×	1.5×	90.1%	1.2×	1.3×
All-Class-Pruned	81.4%	1.2×	1.7×	57.7%	1.2×	1.7×	69.6%	1.2×	1.7×	84.9%	1.4×	2.0×
FAST [57]	90.1%	1.1×	1.2×	64.9%	1.0×	1.0×	74.0%	1.0×	1.0×	91.0%	1.0×	1.0×
PALLEON [18]	90.7%	1.2×	1.3×	65.7%	1.0×	1.0×	74.1%	1.0×	1.0×	92.2%	1.0×	1.1×
CACTUS	**92.2%**	**3.5×**	**5.0×**	**65.4%**	**1.6×**	**2.0×**	**73.8%**	**2.1×**	**2.8×**	**95.9%**	**2.0×**	**2.8×**

Table 1: End-To-End performance of our approach vs baselines on three datasets; Context changes every 30 frames on average. For CACTUS, we used the hybrid switching policy with m-class contexts where $m \in \{2, 3, 4\}$. Speedup values are rounded. The gains improve if wireless data rates are higher (we use 3Mbps) or if caching is used (we assume no caching of the μClassifier heads). Due to space constraints, results for STL-10 are not included but they follow similar trends.



(a) **(b)**

Figure 7: (a) The accuracy of a representative configuration correlates well with the inter-class similarity. (b) The computational complexity of the predicted μClassifiers correlates well with the inter-class similarity. Similarity is normalized to [0, 1] in both figures.

example, using a 40 GPU cluster for training 3-class μCs on the PLACES dataset, our method (similarity-directed sampling) is faster by 2 hours for 20 classes and 270 hours for 100 classes. (b) **CACTUS (Highest FLOPs)** μ**C** employs the μClassifier with the same classes as **CACTUS** but at full image resolution and without pruning (and hence computationally the most expensive). (c) **All-Class Model**, which uses the well-trained EfficientNet-B0 with all the classes and full image resolution. We omit results for other datasets for limited space. Both the classification accuracy and FLOPs are averaged over μClassifiers for all 3-class combinations.

Overall, we see that **CACTUS** can cover a wide trade-off region between accuracy and computational cost — accuracies vary by 3.5% to 5% and compute by 6.5× to 13×. Across the range, the accuracy is nearly as good as the **CACTUS** (Oracle), indicating that *the kNN together with the inter-class similarity metric works well in selecting the best configuration.* In contrast, All-Class Model (red star) and **CACTUS** (Highest FLOPs) (green star) are point solutions. We see that **CACTUS** (Highest FLOPs) has significantly higher accuracy than the All-Class Model even though both models have the same FLOPs.

Effectiveness of the inter-class similarity metric: The good predictive performance of the configuration predictor results from the effectiveness of the inter-class similarity metric in capturing the difficulty level of contexts.

Figure 7a illustrates how the accuracy of a representative μClassifier on each 3-class combination correlates with the inter-class similarity averaged over the 3 classes (Eq. 1). For the graph, all 3-class combinations were binned into six clusters based on the inter-class similarity metric and the accuracy and similarity values are averaged in each cluster. We observed that there is a clear inverse relation between inter-class similarity and classification accuracy. The Pearson correlation between the inter-class similarity metric and the classification accuracy is extremely high (0.86 − 0.97), indicating that *our similarity metric closely mirrors the difficulty level of 3-class combinations.*

Figure 7b looks at this relation from another performance dimension — computation savings. The same procedure of grouping 3-class combinations into six bins is applied here. The Y-axis shows the computation saving of the predicted configurations (by kNN) compared to the All-Class Model/ **CACTUS** (Highest FLOPs) and the X-axis is the inter-class similarity. We see the computation saving drops as the similarity increases since a more powerful model should be used to meet the target accuracy threshold. It also shows that the similarity-aware kNN works well in selecting lightweight configurations for low similarity combinations and more powerful configurations for harder cases.

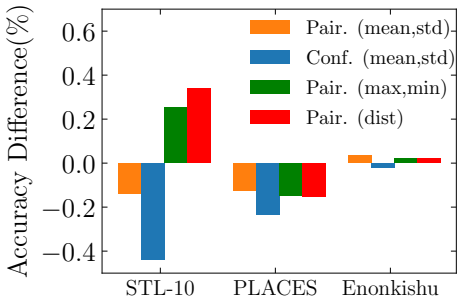


Figure 8: Accuracy difference between the predicted μCs by kNN and oracle μCs using different similarity metrics. Our proposed similarity metric "Pair. (mean, std)" provides the most accurate predictions. Negative difference means the accuracy of the predicted μCs is lower than the oracle.

Comparison with other similarity metrics: We also investigate alternative similarity metrics and context representations including (1) confusion matrix-based similarity, (2) maximum and minimum of pair-wise similarities in a combination, and (3) sorted pair-wise similarities in a combination (distribution-based representation). To compute the pair-wise similarity using the confusion matrix, the number of samples that are misclassified are summed and

normalized by the total number of those two classes. Figure 8 shows the comparison using three datasets. Each bar reflects the accuracy difference between predicted μClassifiers and oracle μClassifiers averaged over all 4-class combinations for each similarity metric. Overall we see kNN using our pair-wise similarity metric "Pair. (mean,std)" provides the most accurate predictions on the STL-10 and PLACES20 datasets compared to other methods. Meanwhile, performance on the Enonkishu dataset matches that of these alternatives.

6.4 Perf. of Context-Aware Switching

This section evaluates the performance of the *context-aware switching* module. We first evaluate the performance of the context change detector and then the hybrid context switching policy.

Effectiveness of the context change detector: A false positive (FP) that wrongly predicts a context change would introduce additional cost whereas a false negative (FN) will cause accuracy degradation. To evaluate the performance of the regression head, we test the μClassifier corresponding to a 3-class combination with the test samples of 3 classes to calculate the FP rate. In order to compute the FN rate, we test the mentioned μClassifier with samples of all out-of-context classes. For each approach, the reported FP and FN rates are averaged over all combinations of 3 classes.

Method	Enonkishu		STL-10		PLACES	
	FP	FN	FP	FN	FP	FN
μC+regression (ours)	3.88%	13.61%	13.58%	10.91%	8.91%	8.18%
μC+maximum prob [24]	16.96%	22.89%	31.44%	33.25%	19.49%	18.29%
Oracle μC+regression	4.01%	13.73%	13.83%	11.15%	8.57%	7.64%

Table 2: False Positive and False Negative rates of our approach vs baselines.

Table 2 compares the context-change detection performance of **CACTUS** ("μC + regression") against baselines: (a) the same μClassifier as **CACTUS** but with a different approach for context change detection based on maximum softmax probability [24]. (b) the **CACTUS** (Oracle) μClassifier with regression head ("Oracle μC + regression"). **CACTUS** is much more accurate at context change detection compared to the baseline that uses the maximum softmax probability. The FP and FN rates of our approach are $10-20\%$ lower than the baseline. **CACTUS** is quite close to the Oracle μClassifer as well. We note that the FP and FN rates can be lowered by increasing the accuracy threshold explained in §6.1 or by using larger μClassifiers (e.g. 4 or 5-class) which would incur fewer context switches. Also, FP and FN rates can be tuned via the context change detection threshold.

Effectiveness of hybrid context switching: We now compare hybrid context switching (i.e. switching between 2-class, 3-class, 4-class, and 5-class contexts) against a fixed-class switching policy (e.g. only 2-class or 3-class or 4-class context). Table 3 shows the accuracy and overall speedup for the Serengeti dataset. We see that for fixed-class switching, accuracy increases as the number of classes increases since the model is less likely to make mistakes when deciding whether to trigger the all-class model. But speedup drops since the models chosen are larger. **CACTUS** (Hybrid) gets the best of both worlds — by intelligently switching, it achieves high accuracy but with more speedup than the fixed-class models.

Methods	Serengeti		
	Acc	SpeedUp(Pi0)	SpeedUp(GAP8)
CACTUS-Hybrid (2,3)	73.6%	1.9×	2.5×
CACTUS-Hybrid (2,3,4)	73.8%	2.1×	2.8×
CACTUS-Hybrid (2,3,4,5)	73.8%	2.1×	2.9×
CACTUS (2-Class)	72.8%	1.4×	1.7×
CACTUS (3-Class)	74.1%	1.3×	1.4×
CACTUS (4-Class)	74.9%	1.0×	1.1×

Table 3: Comparison between the hybrid switching policy versus fixed-class switching

Methods		Enonkishu		Serengeti	
	Storage	Acc	SpeedUp	Acc	SpeedUp
All-Class Model	4.8MB	88.9%	1.0×	74.3%	1.0×
All-Class-Early Exit	4.9MB	84.6%	2.0×	68.7%	1.5×
All-Class-Pruned	2.6MB	81.4%	1.7×	69.6%	1.7×
CACTUS (15 μCs)	19.9MB	89.5%	2.2×	73.6%	2.0×
CACTUS (20 μCs)	20.8MB	90.1%	3.5×	73.9%	2.2×
CACTUS (30 μCs)	23.3MB	90.2%	4.1×	74.0%	2.4×

Table 4: End-to-End performance of Local-Only CACTUS and baselines. Speedup numbers are computed for GAP8 (Rasp. Pi0 results show a similar trend).

Table 3 also shows that the performance of **CACTUS** (Hybrid) generally improves as the number of context sizes i.e. m increases from $m \in \{2\}$ to $m \in \{2,3,4,5\}$. But we get diminishing returns after $m \in \{2,3,4\}$, so that is our sweet spot.

6.5 Perf. of Local-Only CACTUS

We now consider unattended operation (i.e. local-only) — this is the scenario where communication to the cloud is not available and the IoT device uses only locally stored μClassifiers. Since the all-class model also needs to be stored on the IoT device, we choose All-Class EfficientNet for this purpose since it is resource optimized.

In order to decide which μClassifiers should be stored on the device, we used the training set and cached the most frequently used μClassifiers on the device. For local switching, if none of the chosen m-class μClassifier ($m \in \{2,3,4\}$) is available on-device, we simply select an available one that has the highest overlap in classes with the chosen one.

Table 4 shows the accuracy and speedup on GAP8 for **CACTUS** and baselines. We evaluated **CACTUS** with 15, 20, and 30 stored μClassifiers. Results show that by only storing 30 μClassifiers, **CACTUS** can achieve 4.1× and 2.4× speedup for Enonkishu and Serengeti datasets respectively while the accuracy is on a par with baselines. Note that 30 μClassifiers constitute less than 1 percent of all possible 2, 3, and 4-class μClassifiers for the Serengeti dataset. Thus, even with a limited number of μClassifiers and storage **CACTUS** can maintain its advantage in speedup without sacrificing accuracy. **CACTUS** requires more Flash memory than baseline methods, primarily due to the storage of multiple feature extractors. This is, however, not an issue because IoT devices are usually more constrained by RAM instead of Flash memory. For example, Raspberry Pi0 W has 512MB RAM but it supports SD cards for flash.

Metrics	Context Interval						
	Shuffled	3	4	10	30	100	No-CC
Accuracy Gain	-1.3%	0.6%	1.4%	4.1%	5.4%	4.7%	4.0%
Speedup (Pi0)	0.81×	0.94×	1.07×	1.56×	1.96×	1.96×	1.91×
Speedup (GAP8)	1.64×	1.72×	1.97×	2.59×	2.81×	2.59×	2.20×

Table 5: CACTUS accuracy gain and speedup (Pi0 and GAP8) with different context intervals compared to the All-Class Model on PLACES dataset. (No-CC: No-Context-Change)

Metrics	Number of Classes						
	10	20	30	40	60	70	100
Accuracy Gain	4.4%	4.7%	4.3%	4.6%	5.3%	5.6%	6.5%
SpeedUp (GAP8)	2.33×	2.26×	2.18×	2.10×	1.96×	1.86×	1.63×
$SD_{sampling}$ ratio	50.0%	17.5%	10.0%	7.5%	6.0%	5.0%	5.0%

Table 6: Effect of scaling the number of covered classes on CACTUS's performance (PLACES dataset). $SD_{sampling}$ is the ratio of μCs used for training the kNN.

6.6 Ablation Study

Effect of context change rate: We now evaluate how our end-to-end results change as we vary the interval between context changes. We vary the rate from one context change every 100 frames to once every 3 frames on the GAP8 and Pi0 platforms using the PLACES20 dataset. We consider 3-class contexts for synthesizing the temporal patterns. We also include extreme cases such as a) no-context-change where the context is completely static, and b) shuffled where the dataset is shuffled and context changes are extremely frequent since there is no specific temporal pattern. For the no-context-change case, we randomly pick a context (3-class) and evaluate **CACTUS** on it. Experiments were repeated 20 times and the averaged numbers were reported.

Table 5 shows that **CACTUS** maintains its advantage even with higher context change rates i.e. shorter context intervals. The speedup decreases slightly from 2.59–1.96× to 1.72–0.94× and accuracy gain decreases from 4.7% to 0.6% as the interval becomes smaller but **CACTUS** is still considerably better than the baseline (All-Class Model). As the context interval narrows, the all-class model processes an increased number of images, diminishing the accuracy benefits of μClassifiers. Specifically, with context intervals of 3 and 100, the all-class model handles 67% and 11.5% of images respectively. We can see **CACTUS** remains effective even when there is no context change. **CACTUS**, while less accurate than the baseline in shuffled scenarios, maintains a latency advantage on the GAP8 platform but is slower on Pi0.

Scalability: We now look at the scalability of **CACTUS** with increasing the number of classes. Table 6 shows how **CACTUS** end-to-end performance changes relative to the All-Class Model. We used a fixed 3-class switching policy since it was computationally intensive to train 2, 3, and 4-class μClassifiers for large number of classes. We see that **CACTUS**'s accuracy gain over the All-Class Model increases as we scale up the number classes, and its speedup reduces but it is still better. We expect higher speedup if the hybrid policy is utilized. We also see the benefits of similarity-directed sampling with more classes. For example, we only need 5% of the μClassifiers to train the kNN when the number of classes is higher than 60.

Overhead of model training: We briefly report the training overhead for classification and context change detection heads. Since the feature extractor is pre-trained offline, we only need to train the μClassifier's heads (FC layers) which is very fast. To minimize training overhead even more, the embeddings that are input to the FC layers are precomputed. The training overhead for Enonkishu, Camdeboo, Serengeti, and PLACES datasets are on average 5.4, 5.5, 5.7, and 3 seconds respectively on a Tesla M40 24GB GPU.

Figure 9: Raspberry Pi Zero W with a camera (mounted on the Raspberry Pi board), LCD display, and a power bank.

6.7 TrailCam Implementation

We have developed an end-to-end implementation of an "unattended mode" wild-life camera that can capture images of animals or birds and classify them in real-time on the Raspberry Pi Zero W. In this implementation, we store 15 μClassifiers heads and a total of four feature extractors plus an All-Class EfficientNet-B0 for the context prediction. The 15 stored μClassifiers include 7 different configurations. Figure 9 shows our implementation in operation. In the figure, we re-run images from the Camera Trap dataset Enonkishu [6] to illustrate how the system works but we can also directly obtain input from the Raspberry Pi camera.

Figure 10a shows the latency breakdown of key components that contribute to inference – that is, feature extractors corresponding to different image resolutions (7 configurations), μClassifier heads for different contexts, and the All-Class model. Also, we show the latency overhead for context switch i.e. the elapsed time for loading the feature extractor (configuration) for the new context. The results show that the latency for executing μClassifier heads and the context switch overhead is 2 and 1 milliseconds respectively, which is negligible compared to other components. The latency for the seven feature extractor configurations have a roughly 8× dynamic range (94 to 759 milliseconds) which allows **CACTUS** to adapt to the level of difficulty of the current context.

Figure 10b shows the energy breakdown for executing these components. For each component, we break down the energy consumed into "System Overhead" i.e. the energy consumed by the board and peripherals when no computation is being performed

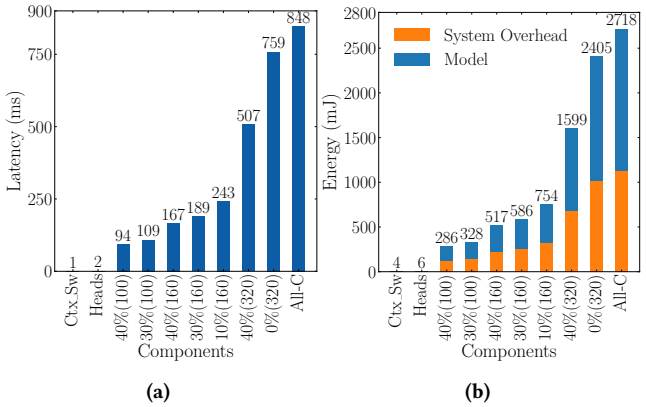


Figure 10: (a) The breakdown of the latency for end-to-end implementation of local-only CACTUS on Pi0. (b) The breakdown of the energy for end-to-end implementation of local-only CACTUS on Pi0. Blue bars correspond to the consumed energy for the execution of the models. Orange bars show the energy consumed by the setup when it is On but not doing any operation. Ctx_Sw and All-C correspond to the context switch and all-class model. μClassifiers are labeled by their configuration parameters. The first number is the pruning ratio and the input size is in the parenthesis.

and "Model" energy consumption which is the energy for actually executing the model. Note that the system overhead is different for different components since the execution time differs. The energy for the system overhead is the power for the board and peripherals multiplied with execution time. The energy for the Model is calculated by subtracting the System Overhead energy from the overall measured energy for each component. The results show that different configurations provide an 8.4× dynamic range in energy consumption which is consistent with the latency range. We also see the consumed energy for head execution and context switch is very low. This validates the fact that reducing the inference time on IoT devices proportionally reduces energy consumption. Hence, even if the application does not require low inference latency, it is important to have an efficient pipeline so that the battery on the IoT device lasts longer.

7 RELATED WORK

Deep Learning in low-power devices: Two popular approaches for the execution of Deep Neural Networks (DNNs) on resource-constrained IoT devices are early exit [37, 59] and model Compression techniques such as quantization [17, 22, 29], model pruning [29, 31, 45, 49], and knowledge distillation [25, 44]. FLEET [27] integrates early exits with computation offloading to further adapt DNN execution for constrained environments. These techniques are either generic or complementary and usually can be combined with other approaches. In this work, we show that both model compression and early exit can be applied to μClassifiers to improve the computation efficiency.

Another line of work is partitioned execution, where the IoT device executes a few layers of a model and offloads the remaining layers to the cloud. Some approaches [36, 60] focus on locating the best partitioning point while others [14, 16, 28, 30] pay attention to intermediate features. However, this approach intrinsically requires continuous wireless communication that limits it to some scenarios.

Efficient video processing: Exploiting the temporal locality of videos is the complementary direction for efficient IoT inference. Prior works fall into two categories. The first set of approaches [26, 33, 35] are query-based and assume scenarios are known offline (static context). The second category of approaches [18, 21, 57] consider dynamic context and try to adapt the deployed models. However, [57] assumes class skews are known and [18, 21] require network connectivity which limits them to specific scenarios. Also, [18, 57] are only effective for applications with long context intervals.

Uncertainty estimation: Our work is also related to uncertainty estimation since detecting context changes is essentially estimating the prediction uncertainty. There are multiple approaches for uncertainty estimation such as Ensemble methods [40], Dropout inference (Bayesian Approximations) [19], Maximum Softmax Probability [24], Test-Time Augmentation [9, 56], and other deterministic methods [20, 50, 51, 53–55]. Most existing methods involve substantial overheads. For instance, Monte Carlo Dropout [19] requires multiple forward passes for uncertainty estimation. Maximum Softmax Probability emerges as the most cost-effective approach for identifying out-of-distribution samples. The maximum value among outputted softmax probabilities for in-distribution samples tends to be larger than for out-of-distribution samples. This approach is inexpensive in terms of computation and memory as it doesn't need any specialized model and it only uses the classification softmax probabilities. However, it did not perform well in our experiments, hence we introduced a regression-based detector to meet our needs.

8 CONCLUSION

This work presents a new paradigm, switchable μClassifiers, to the growing set of approaches for squeezing deep learning models on resource-constrained platforms. We showed that this method can improve accuracy while significantly lowering latency and therefore power consumption. Our work opens up a new direction and can spur follow-on work to fully explore the design space. While we focused on IoT devices in this work, context-aware inference may have broader applicability as models get larger and more complex, for example, in self-driving vehicles.

ACKNOWLEDGMENTS

The research reported in this paper was sponsored in part by the CCDC Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196 (ARL IoBT CRA), and in part by the NSF under Grant No. CNS-2312396, CNS-2338512, CNS-2224054, and DMS-2220211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, NSF or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. We thank our shepherd, Dr. Montanari, for helpful guidance on this paper.

REFERENCES

[1] https://www.tensorflow.org/lite/performance/post_training_quant.

[2] Arm11. https://developer.arm.com/documentation/ddi0360/f/introduction/about-the-processor.

[3] Gap8 tool chain. https://greenwaves-technologies.com/tools-and-software/.

[4] Joulescope js110. https://www.joulescope.com/products/joulescope-precision-dc-energy-analyzer.

[5] Snapshot camdeboo. https://lila.science/datasets/snapshot-camdeboo.

[6] Snapshot enonkishu. https://lila.science/datasets/snapshot-enonkishu.

[7] Snapshot serengeti. https://lila.science/datasets/snapshot-serengeti.

[8] Tensorflow lite benchmark tools. https://www.tensorflow.org/lite/performance/measurement.

[9] Murat Seckin Ayhan and Philipp Berens. Test-time data augmentation for estimation of heteroscedastic aleatoric uncertainty in deep neural networks. In *Medical Imaging with Deep Learning*, 2018.

[10] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning.

[11] Sourav Bhattacharya and Nicholas D Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189. ACM, 2016.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[13] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[14] Hyomin Choi and Ivan V Bajić. Deep feature compression for collaborative object detection. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3743–3747. IEEE, 2018.

[15] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 215–223, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[16] Robert A Cohen, Hyomin Choi, and Ivan V Bajić. Lightweight compression of neural network feature tensors for collaborative intelligence. In *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2020.

[17] Miguel de Prado, Manuele Rusci, Romain Donze, Alessandro Capotondi, Serge Monnerat, Luca Benini, and Nuria Pazos. Robustifying the deployment of tinyML models for autonomous mini-vehicles. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, may 2021.

[18] Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, and Yufei Ding. Palleon: A runtime system for efficient video processing toward dynamic class skew. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 427–441. USENIX Association, July 2021.

[19] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, New York, USA, 20–22 Jun 2016. PMLR.

[20] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna M. Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. A survey of uncertainty in deep neural networks. *CoRR*, abs/2107.03342, 2021.

[21] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136, 2016.

[22] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[24] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. 2016.

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[26] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, Carlsbad, CA, October 2018. USENIX Association.

[27] Jin Huang, Deepak Ganesan, and Hui Guan. Re-thinking computation offload for efficient inference on iot devices with duty-cycled radios. In *The 29th International Conference on Mobile Computing and Networking (MobiCom '23)*, 2023.

[28] Jin Huang, Colin Samplawski, Deepak Ganesan, Benjamin Marlin, and Heesung Kwon. Clio: Enabling automatic compilation of deep learning pipelines across iot and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2020.

[29] Dina Hussein, Dina Ibrahim, and Norah Alajlan. Tinyml: Enabling of inference deep learning models on ultra-low-power iot edge devices for ai applications. *Micromachines*, 13:851, 05 2022.

[30] Sohei Itahara, Takayuki Nishio, and Koji Yamamoto. Packet-loss-tolerant split inference for delay-sensitive deep learning in lossy wireless networks. *arXiv preprint arXiv:2104.13629*, 2021.

[31] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[32] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

[33] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 253–266, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529*, 2017.

[35] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, aug 2017.

[36] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 615–629. ACM, 2017.

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

[38] Liangzhen Lai and Naveen Suda. Enabling deep learning at the lot edge. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2018.

[39] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[40] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6405–6416, Red Hook, NY, USA, 2017. Curran Associates Inc.

[41] Nicholas D Lane, Petko Georgiev, and Lorena Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015.

[42] Ilias Leontiadis, Stefanos Laskaridis, Stylianos I Venieris, and Nicholas D Lane. It's always personal: Using early exits for efficient on-device cnn personalisation. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 15–21, 2021.

[43] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets, 2016.

[44] Jinyu Li, Rui Zhao, Jui-Ting Huang, and Yifan Gong. Learning small-size dnn with output-distribution-based criteria. In *Fifteenth annual conference of the international speech communication association*, 2014.

[45] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning*, pages 5958–5968. PMLR, 2020.

[46] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. 2018.

[47] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3296–3305, 2019.

[48] Arnab Neelim Mazumder, Jian Meng, Hasib-Al Rashid, Utteja Kallakuri, Xin Zhang, Jae-sun Seo, and Tinoosh Mohsenin. A survey on the optimization of neural network accelerators for micro-ai on-device inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021.

[49] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors (Basel, Switzerland)*, 20, 2020.

[50] Marcin Możejko, Mateusz Susik, and Rafał Karczewski. Inhibited softmax for uncertainty estimation in neural networks, 2018.

[51] Jay Nandy, Wynne Hsu, and Mong Li Lee. Towards maximizing the representation gap between in-domain & out-of-distribution examples. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9239–9250. Curran Associates, Inc., 2020.

[52] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104. PMLR, 10–15 Jul 2018.

[53] Maithra Raghu, Katy Blumer, Rory Sayres, Ziad Obermeyer, Bobby Kleinberg, Sendhil Mullainathan, and Jon Kleinberg. Direct uncertainty prediction for medical second opinions. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5281–5290. PMLR, 09–15 Jun 2019.

[54] Tiago Ramalho and Miguel Miranda. Density estimation in representation space to predict model uncertainty, 2019.

[55] Murat Sensoy, Lance Kaplan, and Melih Kandemir. Evidential deep learning to quantify classification uncertainty. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[56] Divya Shanmugam, Davis Blalock, Guha Balakrishnan, and John Guttag. Better aggregation in test-time augmentation. 2020.

[57] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[58] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

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

[60] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017.

[61] https://greenwaves-technologies.com/gap8-product/. GAP8: Ultra-low power, always-on processor for embedded artificial intelligence.

[62] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers.* O'Reilly Media, 2019.

[63] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291. ACM, 2018.

[64] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.