Re-thinking computation offload for efficient inference on IoT devices with duty-cycled radios

Jin Huang, Hui Guan, Deepak Ganesan
University of Massachusetts Amherst, Amherst, MA 01375, USA
{jinhuang,huiguan,dganesan}@cs.umass.edu

ABSTRACT

While a number of recent efforts have explored the use of "cloud offload" to enable deep learning on IoT devices, these have not assumed the use of duty-cycled radios like BLE. We argue that radio duty-cycling significantly diminishes the performance of existing cloud-offload methods. We tackle this problem by leveraging a previously unexplored opportunity to use early-exit offload enhanced with prioritized communication, dynamic pooling, and dynamic fusion of features. We show that our system, **FLEET**, achieves significant benefits in accuracy, latency, and compute budget compared to state-of-art local early exit, remote processing, and model partitioning schemes across a range of DNN models, datasets, and IoT platforms.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.

KEYWORDS

edge computing, cloud computing, computation off-loading, deep neural networks

ACM Reference Format:
Jin Huang, Hui Guan, Deepak Ganesan. 2023. Re-thinking computation offload for efficient inference on IoT devices with duty-cycled radios. In *The 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23), October 2–6, 2023, Madrid, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3570361.3592514

ACM MobiCom '23, October 2–6, 2023, Madrid, Spain

ACM ISBN 978-1-4503-9990-6/23/10…$15.00
https://doi.org/10.1145/3570361.3592514

BLE Specifications	Values
Physical Layer(**PHY**)	125 Kbps, 500 Kbps, 1Mbps, 2Mbps
Connection Interval(**CI**)	7.5ms - 4s(iOS: 15ms min)
Packets per CI(**Packets**)	2 - 8(iOS:4, Android:6)
Packet Size	27 - 251 Bytes
Typical Tx Power	10mW

Table 1: BLE Specifications

1 INTRODUCTION

Recent years have seen substantial interest in deploying Deep Neural Network (DNN) models on resource-constrained Internet of Things (IoT) devices [24, 30, 33, 44]. IoT devices are increasingly interfaced with rich sensors, such as low-power cameras, microphones, and radars and DNNs have shown superior performance on analyzing data from these sensors. It is, however, challenging to execute complex models on these platforms locally due to the high computation demands of DNNs [13, 36].

A number of recent efforts have explored the use of "cloud offload" as a strategy to allow IoT devices to execute DNNs efficiently [17, 20, 22, 25, 37]. These methods typically involve either offloading raw data after compression (e.g. using JPEG or other lossy compression method [32]) or offloading some intermediate results after initial processing at the IoT device (e.g. transmitted compressed intermediate results of a DNN [17, 20]).

However, this body of research has assumed a radio that can transmit whenever needed and as much data as desired. This assumption is at odds with duty-cycled radios which are common in IoT devices. Since the dominant duty-cycled radio is Bluetooth Low Energy (BLE), we focus on that radio in this paper.

A number of BLE parameters can impact the performance of a cloud offload method (shown in Table 1). A key metric is the *connection interval (CI)* i.e. the sleep duration between transmissions which can range from 7.5ms to 4s (longer CIs are preferred to extend battery life). BLE also has relatively low bitrate of under 2Mbps, with lower bitrates if the devices are not proximate to one another. In addition, the number of packets transmitted after each sleep period are also limited. The number varies across platforms and BLE standards — for example, iOS limits BLE 5.0 transmissions to be four packets, and Android limits it to 6 packets[3]. Each of these packets

can be at most 251 bytes. These limitations constrain how fast data can be offloaded to the cloud — even if the connection interval is set to the smallest value, offloading is impractical for BLE-based devices. When reliable transmission is required, BLE transmits the data in one burst, and receives the acknowledgements in the next burst, thus data transfer only occurs every two sleep-wake cycles and average bandwidth is halved. Finally, BLE also can have high packet losses, particularly when larger packet sizes of more than 100 bytes are used and when transmission is frequent [41].

All of these factors *stretch* the transmission of data across several short bursts of data transmission interspersed by long idle times, resulting in inefficient cloud offload. This motivates us to develop a new cloud-offload method that can cooperate with duty-cycled transmissions and reduce inference latency.

Our contribution: This paper proposes a cloud-offload pipeline called **FLEET**[1] that is tailored to the duty-cycling behavior of IoT radios. The basic idea of **FLEET** is to compute locally during the periods when the radio is asleep and yet leverage the cloud using data offloaded during the windows of time when it can transmit data. It intertwines model execution and communication by developing an *early exit model* to mask the latency of using the radio. In the early exit model, a DNN model is enhanced to have cloud-offloadable early exit modules at some layers such that the model only executes until it is sufficiently confident about the result and can exit without having to run the remaining layers. While early-exit models have been extensively studied [21, 39], there are two unique challenges when re-purposing them to enable cloud-offload using duty-cycled radios.

The first challenge is how to prioritize what to transmit during each burst and build in robustness to packet losses. Duty-cycled radios like BLE involve an initial negotiation with the master device (IoT devices are typically the slave), which gets the final say in deciding the sleep-wake parameters such as sleep duration, number of packets per burst, and bandwidth setting. Thus, any model that we design has to be agnostic of these parameters such that it can work across different settings. In addition, IoT radios transmit at very low power for energy-efficiency (e.g. BLE Tx power is typically 10mW whereas WiFi Tx power is typically 1-4W). Hence they are particularly susceptible to packet losses.

In order to utilize limited communication opportunities most effectively, **FLEET** features a *dynamic pooling based prioritized communication* approach that can transmit the most up-to-date intermediate results within the allocated transmission window. Our approach *progressively* transmits data from small to high resolution with feature sharing across these resolutions by using dynamic pooling. This method allows us to be agnostic of radio settings and packet losses and opportunistically transmit as much as possible. It also allows us to stop transmitting intermediate results from a previous layer when the current layer finishes execution and seamlessly switch to the next layer's results.

The second challenge is how to handle the limited features transmitted to the cloud. Offloading early exit computation to the cloud has the advantage of being able to use a lot more resources to execute deeper models. However, it still needs to contend with the fact that only limited data is available per burst. The cloud needs to leverage all available information, i.e. all data that is transmitted by the IoT device from the previous bursts, not just the data from the current burst. This, in turn, requires us to develop a model that can leverage variable-sized intermediate results transmitted from prior layers to improve task accuracy.

In order to fuse variable amounts of data from different layers, **FLEET** proposes an server-side *feature fusion* approach that allows the server to take advantage of data transmitted during the execution of different layers, even if they are not at the same resolution or have the same number of intermediate results. We use up-sampling and dense blocks for feature fusion and thereby reduce the cost of training.

We evaluate **FLEET** extensively across the range of BLE parameter settings, three datasets and multiple IoT processors, and show that:

- Compared to local execution, **FLEET** can achieve 2.3× – 4.7× speed-up in latency and 2.1× – 4.4× saving in energy. Compared to Edge-cloud Model Partitioning, **FLEET** achieves 1.2× – 3.3× speed-up in latency and 1.8× – 2.6× reduction in energy. Compared with JPEG-compressed data offload, **FLEET** achieves 1.7× – 6.1× speed-up in latency and 1.6× – 4.4× energy savings.
- Overall, **FLEET** can achieve 1.2×–4.0× reduction in latency when compared with the best of all local and remote execution baselines; **FLEET** also achieves 1.2×–2.5× saving in energy consumption.
- **FLEET** outperforms baselines on a range of IoT processors (from low end Cortex-M33 [2] to more powerful GAP8 [42] and Cortex-A77 [1]), across different datasets (ImageNet-100, TinyImageNet and CIFAR-100), and different models (MobileNetV3, ResNet34 and InceptionV3).
- **FLEET** is robust to packet losses and can operate seamlessly without requiring knowledge of underlying duty-cycling parameters while providing substantial latency speed-up.

2 CASE FOR FLEET

In this section, we argue that cloud offload of early exits presents new opportunities for IoT devices with duty-cycled

[1] FLEET = oFLoad Early ExiT

radios such as Bluetooth Low Energy (BLE). We begin with a brief background on BLE.

BLE is a successor to Bluetooth Classic that is specifically designed for highly constrained IoT devices. While BLE can be used for unscheduled data transfer, this is not ideal for extending battery lifetime. As described earlier, the most efficient mode of operating a BLE radio is in duty cycled mode, i.e. the radio sleeps for tens or hundreds of milliseconds and wakes up to transmit a few packets before turning off its radio. In this mode, BLE consumes less than 10% of the energy of Bluetooth Classic.

While duty-cycling benefits energy-efficiency, it adversely affects latency for any cloud-offload method. Thus, canonical methods such as compressing images via JPEG and transmitting to the cloud incurs hundreds of milliseconds of delay due to the data being chopped up into small bursts that fit in each connection interval.

Our core idea to address the problem is cloud-based early exit. We refer to a DNN model that has early exits as an *early-exit model*, the DNN model itself as the *base model*, and the attached layers for an early exit as a *early-exit module*. Cloud-based early exit offloads the computation of early-exit modules to the cloud. It allows the base model to be executed locally on device during the periods when the radio is asleep and early exit modules on the cloud to continue the computation on intermediate data offloaded when the radio is active. There are three high-level reasons why we expect this architecture to be more naturally aligned with the needs of both cloud offload and the constraints of duty-cycled radios.

Pipelining cloud-offload and local execution: The first advantage of cloud-based early exit is the ability to use a duty-cycled radio and processor in a pipelined manner to mask the latency of using the radio. This addresses a limitation of canonical cloud-offload methods, for example, transmission of JPEG-compressed data or edge-cloud model partitioning, which are blocking i.e. they are idle during radio sleep cycles. In contrast, early-exit computation offloaded to the cloud is non-blocking, i.e. the computation of the next stage in the base model can occur simultaneously with the computation of the early-exit module.

Figure 1a shows latency for a model with and without early exit. When there is no early-exit, the latency is fixed for all data cases; when there is early exit, the latency is more evenly spread out with only the hard cases incurring the maximum latency. Since the radio and processor can operate in parallel, the IoT device can transmit part of the intermediate results after each layer to the cloud for early exit computation while the next layer of the base model is executing locally. This allows it to operate in a non-blocking manner and use both local and remote resources to minimize inference latency.

Leveraging deeper early exit modules: The second advantage is the cloud-based early exit allows us to leverage deeper models in the cloud. While early exits can be implemented with fewer layers to execute locally, this sacrifices accuracy compared to deeper models that can execute on the cloud.

Figure 1b compares three versions of a local early-exit model that uses one fully-connected layer (**1 FC**), two convolutional layers and one fully-connected layer (**2 Conv + 1 FC**) and four convolutional layers and one fully-connected layer (**4 Conv + 1 FC**) in its early-exit modules. We see that adding two additional convolutional layer improves accuracy of the early-exit model by 15% for earlier exits and 7% for later exits; and two additional layers improves accuracy by 10% for earlier exits and 4% for later exits.

Thus, we see that additional resources for the early-exit computation is useful and offloading this block to the cloud can allow more accurate early exit computation to improve performance of the base model on the IoT device.

Fusing intermediate results from multiple layers: The third advantage is that early exit modules on the cloud can leverage not just the intermediate results from the layer they are attached to but all prior layers as well. Local early exit on resource-constrained platforms typically work only on intermediate data from the specific layer that they are attached to, but this diminishes accuracy.

Figure 1c compares the three versions of early exit a) using only two channels that were transmitted after the current layer, b) fusing two channels from the current and previous layer, and c) fusing two channels from the previous two layers with the current layer. We see that if we used the channels from prior layers, accuracy improves and is higher than what can be achieved with features from a single layer.

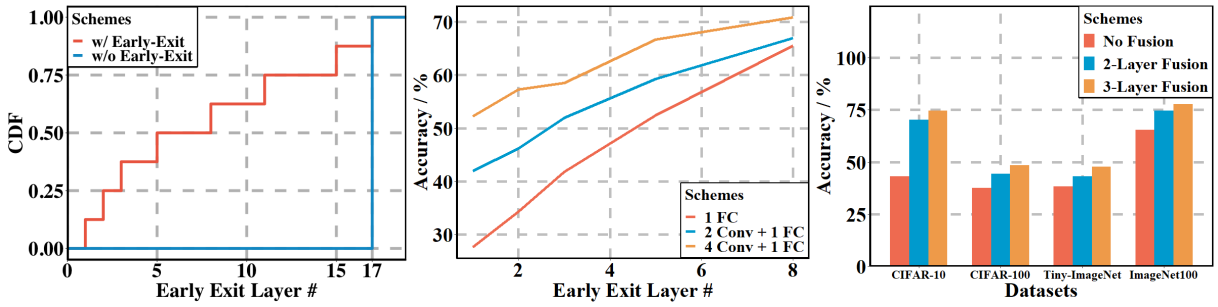
Fusing data from multiple layers is particularly useful for cloud offload since duty-cycled radios cannot transfer a lot of intermediate results after each layer. However, since resources are plentiful at the server, it can fuse all data transmitted from previous layers to offset this drawback and still achieve good performance.

3 DESIGN OF FLEET

3.1 Overview

The **FLEET** pipeline consists of three main components as shown in Figure 2 — *Encoder*, *Fusion Module* and *Cloud Early-Exit*. The cloud early-exit components and the fusion modules form a complete set of early-exit modules.

Encoder: The goal of the encoder is to compactly encode the features of each layer to facilitate transmission in the limited communication opportunities. A key design principle



(a) (b) (c)

Figure 1: Three advantages of early exit for duty-cycled cloud offload. (a) Cumulative Distribution Function (CDF) shows the fraction of data cases that exit early at each layer. In our setup, Early Exit transmits and exits after each layer when the prediction logits is larger than 0.7, which is well aligned with duty-cycled radios and provides a more graded latency profile (b) Cloud offload allows us to fuse features from several previous layers, which also increases accuracy, and (c) Cloud offload allows us to use more layers in the early exit module which increases accuracy.

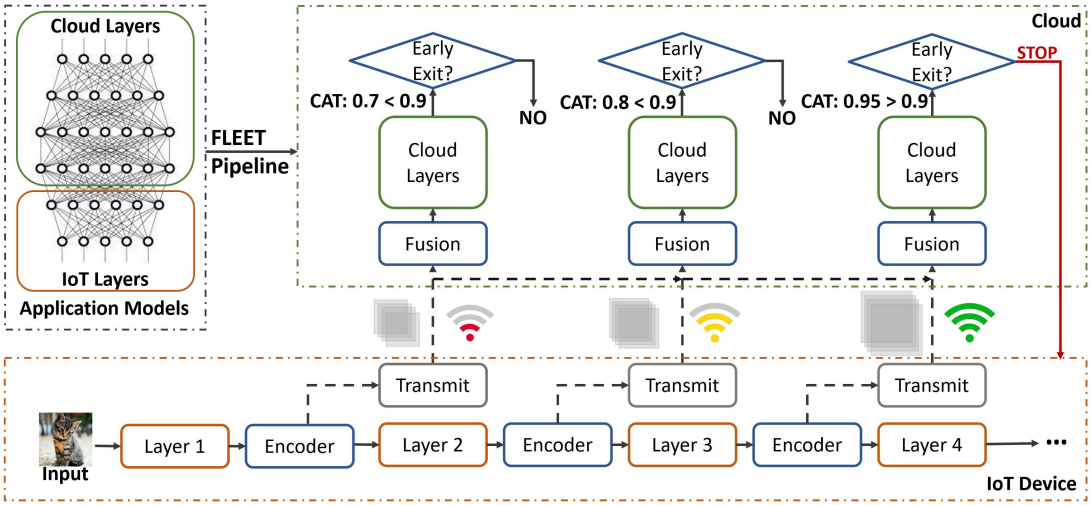


Figure 2: Overall Design of FLEET.

is that it should make minimal assumptions about the current radio duty-cycling parameters and packet loss so that the encoder can work effectively even if these parameters change.

The encoder works as follows. The layers on the IoT devices are sequentially executed as usual. Between each layer, an Encoder layer is inserted that encodes the features of the current layer via *dynamic pooling* and progressively transmits the features during duty-cycling wakeups that occur during the execution of the next layer. Since the encoder performs only lightweight pooling operations, its runtime overhead is negligible.

We assume that the Convolutional layers/blocks to be executed on the IoT devices are $L_{1:N}$. The feature map $\mathbf{F}_{1:N}$ from each layer is a 3-dimensional vector whose size is defined by its width, height and the number of channels(depth). The encoder will reduce the feature map $\mathbf{F}_{1:N}$ into the smaller feature map $\widetilde{\mathbf{F}}_{1:N}$ by reducing both its height/width and depth.

Fusion module: This module fuses the features received by the cloud and produces fused features to early-exit components for prediction. A fusion module aggregates and fuses features of different sizes from different layers. Fused features are better because the cloud can take advantage of all data transmitted so far which leads to better task accuracy.

Cloud early-exit: Finally, the cloud early-exit component generates prediction results. **FLEET** uses the prediction results to determine whether to stop the execution on the IoT devices and returns the results to the IoT device if no further execution is necessary.

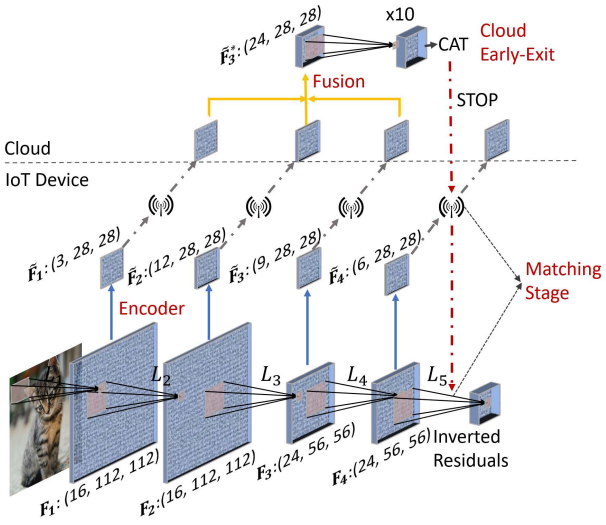


Figure 3: Example: FLEET Pipeline on MobileNetV3. Matching stage means the execution of the Inverted Residual is overlapped with communication.

Example: Figure 3 shows an example of our pipeline with the first five layers of the MobileNetV3 model. $L_{1:5}$ are the different layers of convolutional operations or inverted residual blocks. $\mathbf{F}_{1:4}$ shows the original feature size from each layer, and $\widetilde{\mathbf{F}}_{1:4}$ is the compact encoded feature for transmission. At the cloud, all features available from previous layers are fused and provided as input to the cloud early-exit component. For example, when layer 4 is executing, the previously received features $\widetilde{\mathbf{F}}_{1:3}$ will be fused and used to generate a final prediction. The cloud early-exit component is not limited by compute resources. Hence it is a deeper network than a simple fully-connected layer — ×10 on the cloud early-exit component means there can be a large number of layers used in the cloud to generate final predictions. If the final prediction is sufficiently confident, it will stop the ongoing execution at the IoT device by sending back a notification to the IoT device.

Next, we explain these components in more detail.

3.2 Encoder

The Encoder is designed to encode the transmitted features so that it can fit into the limited transmission opportunities provided by the radio. As mentioned earlier, the Encoder should be agnostic of the current radio duty-cycling parameters and be designed to work irrespective of what is negotiated with the recipient.

FLEET performs *feature size reduction* on the selected channels in multiple tensor dimensions simultaneously and leverages *dynamic pooling* to enable *progressive transmission* in case of BLE packet loss.

Feature Size Reduction: Since duty-cycling restricts the amount of data transmitted by each node, only a subset of features $\mathbf{F_n}$ from layer L_n can be transmitted during the execution time of layer L_{n+1}. Therefore, the encoder needs to reduce the feature sizes to squeeze it into available transmission opportunities.

There are many different ways to achieve this compressed mapping; we look at a specific approach that applies to image-based tasks: the features $\mathbf{F_n}$ can be represented by a three-dimensional tensor (c_n, s_n, s_n), where c_n refers to the number of channels while s_n refers to the height/width from the n-th layer. For simplicity, we assume the height and width is the same size.

Since the radio transmission opportunities are quite limited, it is wasteful to encode all the intermediate results which can be sparse. Therefore, we assume the encoder is designed with a priori idea of the maximum size that should be transmitted. This information is calculated based on the execution time of each layer on the chosen platform and the best-case duty-cycling parameters. For example, if Layer L_2 takes t_2 milliseconds to execute, and the BLE radio setting is 10ms CI and 8 Packets of 251 Bytes per CI, then the maximum data amount that can be transmitted in parallel with Layer L_2's execution is $251 \times 4 \times \lceil t_2/10 \rceil$ bytes. Therefore, our design target is to reduce both width/height and depth of the feature map to fit into this data size without performance degradation. We note that this is not a strict limit, so for example, if t_2 is too small to transmit enough data during connection intervals, we can wait to transmit more data at the cost of extra transmission latency.

Mathematically, the maximum data size $\widetilde{S}_n^{max}$ of layer L_n can be calculated by the execution time t_{n+1} of the next layer L_{n+1} given the BLE radio working at t_{CI} CI and packet size S_{Packet}

$$\widetilde{S}_n^{max} = S_{Packet} \times \lfloor \frac{t_{n+1}}{t_{CI}} \rfloor. \quad (1)$$

We denote the size of feature maps $\widetilde{\mathbf{F}}_\mathbf{n}$ of the maximum data size to be $(\widetilde{c}_n, \widetilde{s}_n, \widetilde{s}_n)$, then we have $\widetilde{c}_n \times \widetilde{s}_n \times \widetilde{s}_n \leq \widetilde{S}_n^{max}$. We can sample a series of smaller feature $\widetilde{\mathbf{F}}_{\mathbf{n,1:M}}$ of the size $(\widetilde{c}_n, \widetilde{s}_{n,1:M}, \widetilde{s}_{n,1:M})$ to fit into the different duty-cycling parameters and packet loss rate.

Progressive Transmission: Progressive transmission is a natural fit to deal with 1) the fact that different duty-cycling parameters may lead to different amounts of data transmitted 2) packet loss can happen during BLE radio transmissions. In this approach, we always encode and transmit the features of smallest size and gradually increase the feature size until we reach the execution latency of the next layer or the maximum feature size. Note that an advantage of progressive transmission is that the IoT device does not need to explicitly synchronize its transmission with its execution — even if

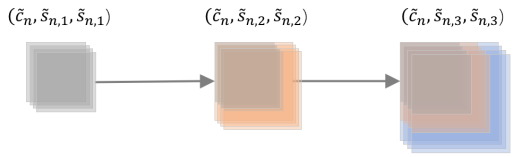


Figure 4: Illustration of Progressive Transmission of the Features from Dynamic Pooling. The feature $\widetilde{\mathbf{F}}_{\mathbf{n},1}$ of the shape $(\widetilde{c}_n, \widetilde{s}_{n,1}, \widetilde{s}_{n,1})$ are part the feature $\widetilde{\mathbf{F}}_{\mathbf{n},2}$ so that we only need to transmit the extra part(the orange part) to reconstruct the feature $\widetilde{\mathbf{F}}_{\mathbf{n},2}$ from $\widetilde{\mathbf{F}}_{\mathbf{n},1}$. Similarly, we only need to transmit the extra data (the blue part) to reconstruct the feature $\widetilde{\mathbf{F}}_{\mathbf{n},3}$ from $\widetilde{\mathbf{F}}_{\mathbf{n},2}$.

the execution of the next layer is finished earlier than transmission of features from the previous layer, the IoT devices can simply discard the previous unsent features and start to send the new features from the next layer. This decoupling prevents communication and computation from blocking on each other.

For example, when we transmit the feature $\mathbf{F_n}$ by possibly encoding it into a series features $\widetilde{\mathbf{F}}_{\mathbf{n},1:\mathbf{M}}$ of different shapes $(\widetilde{c}_n, \widetilde{s}_{n,1:M}, \widetilde{s}_{n,1:M})$, we can always start to encode and transmit from the shape $(\widetilde{c}_n, \widetilde{s}_{n,1}, \widetilde{s}_{n,1})$, and stop at some shape $(\widetilde{c}_n, \widetilde{s}_{n,m}, \widetilde{s}_{n,m})$.

But such a progressive transmission approach still results in extra transmission overhead. For example, originally we only need to transmit the feature $\widetilde{\mathbf{F}}_{\mathbf{n},\mathbf{m}}$ of one shape $(\widetilde{c}_n, \widetilde{s}_{n,m}, \widetilde{s}_{n,m})$; but now we need to first transmit $m-1$ features $\widetilde{\mathbf{F}}_{\mathbf{n},1:\mathbf{m}-1}$ of different shape $(\widetilde{c}_n, \widetilde{s}_{n,1:m-1}, \widetilde{s}_{n,1:m-1})$.

Dynamic Pooling: We propose a novel solution, *dynamic pooling*, to fit the features into the short transmission windows without incurring extra overhead. Dynamic pooling enables the reuse of the features during progressive transmission by special design of the pooling operations. We make the smaller features to be part of the larger feature. When transmitting the larger feature, we don't need to transmit from scratch but only the difference part from the smaller features. Figure 4 illustrates the basic idea of dynamic pooling.

Given the size of input feature map (s_i, s_i) and the size of output feature map (s_o, s_o), we use two different kernels of size k and $\frac{k}{2}$ pooling the feature map where k **mod** $s_i = 0$. We assume that there are respectively m and n values generated by two kernel sizes, then we have:

$$\begin{aligned} k \cdot m + \frac{k}{2} \cdot n &= s_i, \\ m + n &= s_o. \end{aligned} \tag{2}$$

Solving the equation, we have $m = \frac{2 \cdot s_i}{k} - s_o$ and $n = 2 \cdot s_o - \frac{2 \cdot s_i}{k}$. We denote the average pooling using the kernel of size (k, k) as $\mathbf{Pool}_{(k,k)}(\cdot)$. Then the dynamic pooling will pool the feature $\mathbf{F}[0{:}s_i, 0{:}s_i]$ into a pooled feature $\widetilde{\mathbf{F}}[0{:}s_o, 0{:}s_o]$ in the following way without overlap:

$$\begin{aligned} \widetilde{\mathbf{F}}[0{:}n, 0{:}n] &= \mathbf{Pool}_{(k/2,k/2)}(\mathbf{F}[0{:}\frac{kn}{2}, 0{:}\frac{kn}{2}]) \\ \widetilde{\mathbf{F}}[0{:}n, n{:}s_o] &= \mathbf{Pool}_{(k/2,k)}(\mathbf{F}[0{:}\frac{kn}{2}, \frac{kn}{2}{:}s_i]) \\ \widetilde{\mathbf{F}}[n{:}s_o, 0{:}n] &= \mathbf{Pool}_{(k,k/2)}(\mathbf{F}[\frac{kn}{2}{:}s_i, 0{:}\frac{kn}{2}) \\ \widetilde{\mathbf{F}}[n{:}s_o, n{:}s_o] &= \mathbf{Pool}_{(k,k)}(\mathbf{F}[\frac{kn}{2}{:}s_i, \frac{kn}{2}{:}s_i]) \end{aligned} \tag{3}$$

Example of Dynamic Pooling: We now give an visual example of how dynamic pooling works. Given a feature map $\mathbf{F_n}[0{:}s, 0{:}s]$and a kernel (k, k) where k **mod** $s = 0$, we generate a pooled feature map $\widetilde{\mathbf{F}}_\mathbf{n}[0{:}s/k, 0{:}s/k]$ by averaging within the kernel without overlap. The value of the generate feature $\widetilde{\mathbf{F}}_\mathbf{n}$ at the index $(0, 0)$ will be

$$\widetilde{\mathbf{F}}_\mathbf{n}[0, 0] = \mathbf{Pool}_{(k,k)}(F_n[0{:}k, 0{:}k]) \tag{4}$$

If we change the kernel size of the first k column and row from k to $\frac{k}{2}$, but use a kernel of size k for the other columns and rows, we can get a new feature map of $\widetilde{\mathbf{F}}'_\mathbf{n}$ of size $(\frac{s}{k}+1, \frac{s}{k}+1)$. Compared with the original generated feature $\widetilde{\mathbf{F}}_\mathbf{n}$ of size (k, k), we have

$$\widetilde{\mathbf{F}}'_\mathbf{n}[i+1, j+1] = \widetilde{\mathbf{F}}_\mathbf{n}[i, j], \forall 1 \le i, j \le k, \tag{5}$$

which means the smaller features are guaranteed to be part of the larger features. In addition, the re-construction of the new features will also re-use the values from smaller features, for example, $\widetilde{\mathbf{F}}'_\mathbf{n}[0, 0]$, $\widetilde{\mathbf{F}}'_\mathbf{n}[0, 1]$ and $\widetilde{\mathbf{F}}'_\mathbf{n}[1, 0]$ are transmitted while $\widetilde{\mathbf{F}}'_\mathbf{n}[1, 1]$ is re-constructed using $\widetilde{\mathbf{F}}_\mathbf{n}[0, 0]$ on the cloud:

$$\begin{aligned} \widetilde{\mathbf{F}}'_\mathbf{n}[0, 0] &= \mathbf{Pool}_{(k/2,k/2)}(F_n[0{:}k/2, 0{:}k/2]), \\ \widetilde{\mathbf{F}}'_\mathbf{n}[0, 1] &= \mathbf{Pool}_{(k/2,k/2)}(F_n[0{:}k/2, k/2{:}k]), \\ \widetilde{\mathbf{F}}'_\mathbf{n}[1, 0] &= \mathbf{Pool}_{(k/2,k/2)}(F_n[k/2{:}k, 0{:}k/2]), \\ \widetilde{\mathbf{F}}'_\mathbf{n}[1, 1] &= 4 \times \widetilde{\mathbf{F}}_\mathbf{n}[0, 0] - \widetilde{\mathbf{F}}'_\mathbf{n}[0, 0] - \widetilde{\mathbf{F}}'_\mathbf{n}[0, 1] - \widetilde{\mathbf{F}}'_\mathbf{n}[1, 0]. \end{aligned} \tag{6}$$

Figure 5 shows an example how dynamic pooling generate the features from $(\frac{s}{k}, \frac{s}{k})$ to $(\frac{s}{k}+2, \frac{s}{k}+2)$. We can see that the values from the kernel (k, k)(the orange patches) are the same and can be directly reused. The other colors are pooled with kernels of different shapes $(\frac{k}{2}, \frac{k}{2})$, $(\frac{k}{2}, k)$ and $(k, \frac{k}{2})$. The position information is already hard encoded when generating the features so we have no extra overhead to transmit any position information.

3.3 Dynamic Feature Fusion

Figure 6 shows the design of the fusion module in the **FLEET** pipeline. Compared with an approach that inference using the current layer feature, feature fusion can lead to better task accuracy by leveraging information from previous layers. In **FLEET**, the features to be fused might be of different

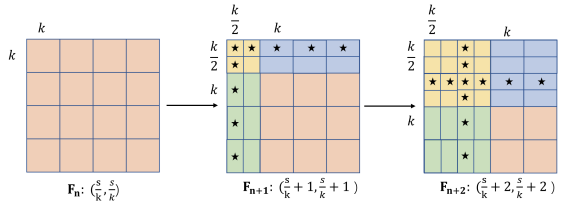


Figure 5: Example of dynamic pooling to generate features from $(\frac{s}{k}, \frac{s}{k})$ to $(\frac{s}{k}+2, \frac{s}{k}+2)$. The asterisk means the average value pooled from the block will be transmitted.

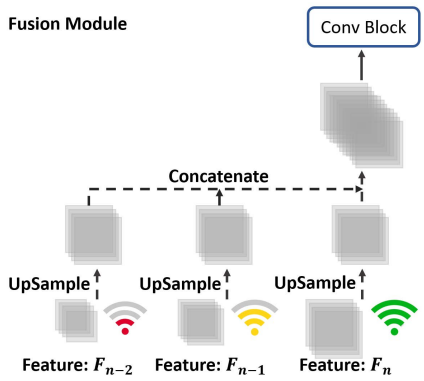


Figure 6: Overall design of the fusion module.

sizes or dimensions. Assume that we are fusing k features $[\widetilde{\mathbf{F}}_{\mathbf{n-k+1}}, ..., \widetilde{\mathbf{F}}_{\mathbf{n}}]$ of different sizes. We first use up-sampling operations to unify the spatial sizes of different features and then the concatenation operation to merge these feature tensors into one feature tensor.

$$\widetilde{\mathbf{F}}'_{\mathbf{i}} = \mathbf{UpSample}(\widetilde{\mathbf{F}}_{\mathbf{i}}), i \in [n-k-1, n],$$
$$\widetilde{\mathbf{F}}^*_{\mathbf{n}} = \mathbf{Concat}([\widetilde{\mathbf{F}}'_{\mathbf{n-k+1}}, ..., \widetilde{\mathbf{F}}'_{\mathbf{n}}])). \quad (7)$$

Finally a convolutional operation f_n will be applied so that the output channels are the same as the input channels of the layers on the cloud.

We denote the spatial sizes as a set S and the combinations of spatial sizes from k layers are S^k. Because the feature distribution of different spatial sizes will be different, it is preferable to use a specific convolutional block for each combination of spatial sizes. However, this leads to an exponential increase in training cost - fusing features from k layers of $|S|$ different spatial sizes can leads to $|S|^k$ combinations and training cost. In our experiments, we select $k = 2$ since this gives sufficiently high accuracy and reduces the combinations and training cost.

The cloud execution is also non-blocking i.e. the cloud model does not need to actively wait for all features to arrive; when the new features are received, the cloud model fuses it with the feature maps from the previous layer to get the new inference results.

3.4 Cloud Early-Exit

Once the features transmitted from previous layers are fused, the cloud continues execution of the base model without modification. We denote the layers $L_{1:n}$ $(n < N)$ on IoT devices as a function $g^{IoT}_{1:n}(*)$ and the cloud layers as a function $g^c(*)$. We denote the inputs and ground truth labels as $\mathbf{X}$ and y.

For radio bandwidth setting b_m and k layer fusion, the loss function for the prediction after the transmission of the intermediate results from the n- layer will be:

$$\begin{aligned}
\text{Encoder: } & \widetilde{\mathbf{F}}_{\mathbf{n}} = e_n(g^{IoT}_{1:n}(\mathbf{X})), \\
\text{Fusion: } & \widetilde{\mathbf{F}}^*_{\mathbf{n}} = f_n(\mathbf{Concat}(\mathbf{UpSample}([\widetilde{\mathbf{F}}_{\mathbf{n-k+1}}, ..., \widetilde{\mathbf{F}}_{\mathbf{n}}]))), \\
\text{Outputs: } & \hat{y}_n = g^c(\widetilde{\mathbf{F}}^*_{\mathbf{n}}), \\
\text{Loss: } & l_n = loss(\hat{y}_n, y), \quad (8)
\end{aligned}$$

where e and f are the encoding and fusion functions, and $loss$ is the loss function.

Early-Exit Trigger: To decide whether to early stop at some layer, the cloud early-exit component uses the confidence or the entropy of the prediction. Given a prediction result $\hat{y}$, the confidence score can be calculated by the top-1 softmax value $\mathbf{softmax}(\hat{y})$. Early stop happens when the confidence score is larger than a predetermined threshold. The cloud early-exit component then sends the exit signal back to IoT device to stop the execution.

We note that a key advantage of cloud early-exit is that the cloud early-exit can execute many convolutional layers for feature extraction before the prediction happens. Thus, even if the features that have been transmitted to the cloud are not complete, the ability to extract better features counters the lack of fidelity in the data.

4 EVALUATION

This section evaluates **FLEET** against state-of-the-art early exit-based models and partitioned execution models under a range of BLE radio duty-cycling settings and IoT platforms. We start by describing the datasets, experimental settings, and baselines that we compare against.

4.1 Experiment settings

Dataset: We evaluate the **FLEET** pipeline primarily on **ImageNet-100** [8], which has 224 × 224 images from 100 different subclasses from ImageNet. The ImageNet size is commonly seen on IoT devices with QVGA or VGA cameras.

For lower-end IoT platforms with too limited memory and compute to process ImageNet images, we also show a few results for the **CIFAR-100** [23] dataset which has 32 ×

32 image inputs from 100 different classes; and from **Tiny-ImageNet** [6] which has 64×64 images with 200 different subclasses from ImageNet.

DNN Models: We implement the pipeline on different backbone models including: MobileNetV3 [15], ResNet34 [12] and InceptionV3 [38]. Since different datasets have different resolutions, we modified the model structure to fit the dataset. For MobileNetV3, we replace two inverted residual blocks (layer 3, 14) of stride 2 with the stride 1 for CIFAR-100; we replace one block (layer 5) of stride 2 with 1 for Tiny-ImageNet; for InceptionV3, we remove two max pooling layers for CIFAR-100; and for Tiny-ImageNet, we remove one max pooling layer.

FLEET Pipeline: For the **Encoder**, we sample different spatial sizes to prioritize transmissions across CIs. For ImageNet, the spatial sizes are from 10×10 to 28×28 with a step size of 2; for CIFAR-100, the spatial sizes are from 3×3 to 8×8; and for Tiny-ImageNet, the spatial sizes are from 6×6 to 16×16 with a step size of 2. The depths of the features to encode after each layer is chosen based on the execution latency of the layer. For example, the execution latency of third layer of MobileNetV3 on CIFAR-100 is roughly 3× of that of the second layer, so the depth of the second layer will be 3× of the first layer.

For **Cloud Early-Exit**, we threshold the confidence scores of the prediction results from each layer to determine whether to early stop at the current stage. The threshold is a hyper-parameter that is profiled once for different datasets. In order to get the best trade-off between compute efficiency and accuracy, we linearly sample the thresholds from 0.1 to 0.9 at a step size 0.1. In our experiments, we use the accuracy on validation set to determine the appropriate threshold for each dataset.

For the datasets we work with, we find that we only need early exits for the first 8 layers of MobileNetV2 and first 5 layers of ResNet34 and InceptionV3 because the performance already converges to the maximum accuracy.

Local Early-Exit Baselines: We compare against three state-of-the-art local early-exit methods: 1) **BranchyNet** [39] adds early exit branches to the existing models and jointly trains them with weighted sum loss from all early-exit modules; 2) **Shallow-Deep** Networks [21] add both early exit modules and a specially designed feature reduction module. It also uses variant coefficients as training proceeds so that the early exits whose performances are poor in early training stage will not interfere with the latter exits; 3) **Ensemble** [9] models are a series of models from the original model but the number of layers are reduced to generate predictions at different compute levels.

For these baselines, we add a series of exit points for the models including layer $1-3$, 5, 8, 11, 13 and the final output. The layers are chosen because they are transition layers where the spatial sizes of features are reduced. For Shallow-Deep network, the weights for the loss function are adaptively tuned by its training process; for BranchyNet, we uses 0.05 for the first 2 layers, 0.1 for the layers in between and 0.4 for the final output to get rid of the interference from the early layers. The pooling layer before the early-exit classification head of BranchyNet and Ensemble models will pool the features to different shapes from 4×4 or 2×2 based on the depth of the feature.

Model Partitioning Baselines: We compare against two IoT-cloud partition-based models as baselines: 1) **CLIO** [17] transmits intermediate features from a partition point in a progressive manner and applies multiple prediction heads on the cloud side to offer different levels of performance; 2) **NeuroSurgeon** is a baseline that is adapted from [20], which also partitions the model. While NeuroSurgeon can adapt the choice of partition point, it cannot adapt to dynamic changes in bandwidth for a given partition point. Hence, we augment it to send features of different sizes by changing the spatial size such that it can adapt to different bandwidth conditions.

We choose the layer 5 to split the model as this is the best partition point for MobileNetV3 [16]. The different depths that we choose for CLIO are from 2 to 28 for MobileNetV3 and the spatial sizes that we choose for NeuroSurgeon are the same as those used in dynamic pooling in **FLEET**.

Metrics: The three main metrics we evaluate are *accuracy*, *inference latency*, and *energy*. Accuracy is the classification accuracy on the selected dataset. Latency and Energy are based on profiling on different typical IoT platforms and radios as described below.

Implementation on IoT Platforms: We select three IoT processors from highly resource limited to more moderately capable – at the lower end, the ARM **Cortex-M33**@64MHz on the nRF5340 MCU and at the upper end, the 32-bit Risc-V **GAP8** neural accelerator [42] @175MHz and the ARM **Cortex-A77** on Raspberry Pi [35] @200MHz (or higher). Based on our profiling, we find that the Cortex-M33 has the least compute ability while the compute capability of GAP8@175MHz lies in between the Cortex-A77@200MHz and Cortex-A77@400MHz.

We implemented our system on the three above platforms and profiled the latency of model execution. We use **TVM** and **MicroTVM** [34] as the backend to compile the deep learning models into executable on different platforms working at different frequency. Then we collect the execution latency of the deep learning models early-exit at the different layers for our evaluation. Due to the fact that the Flash and RAM size on the nRF5340DK is very limited, only

Processor (Platform)	Frequency / MHz	Power / mW	RAM / KByte	Flash / Mbyte
Cortex-M33 (nRF5340DK)	64	9.3	512	1
Risc-V GAP8 (GAP8)	175	16.4	100(L1) 512(L2)	64
Cortex-A77 (Raspberry Pi)	200 - 1500	40 - 750	128(L1) 512(L2) 2GB(RAM)	32GB

Table 2: Platforms

the models with input size 32x32x3 are deployed; for other platforms, models with ImageNet sizes are deployed.

Radios: We evaluate over the full range of duty-cycling parameters offered by BLE shown in Table 1. The maximum payload size of 1 Mbps and 2 Mbps PHY is 251 Bytes while that of 125 Kbps is only 27 Bytes. The connection intervals of BLE radios starts from 7.5ms to 4s at increments of 1.5ms. For the packets per connection interval, we study commonly seen options from 2 to 8 packets per CI including payloads and acknowledgements, which means 1 to 4 payloads.

4.2 Latency advantages of FLEET

Table 3 presents the speed-ups offered by **FLEET** across the above-mentioned range of BLE parameters and IoT platforms. The columns on "Local Early Exit", "Model Partition" and "JPEG" summarize the benefits offered by **FLEET** over these schemes individually, and the column "All" shows the region where **FLEET** is better than all other methods cumulatively. We compare **FLEET** with these baselines around the same accuracy: specifically we compare with baselines at around 88% on ImageNet-100 and 68% on CIFAR-100. For each column, the "CI" sub-column shows the connection interval regime where **FLEET** performs better than other baselines, and the "Speed-up" column shows the latency speedup by using **FLEET**. In addition to considering all the BLE parameters, we also look at the effect of the computational capability of the IoT device. For the Cortex-M33, we use the CIFAR-100 dataset since resource limitations preclude processing of larger images; all other results are for ImageNet-100.

Overall latency speedup: Let us first look at the overall results in the column "All". We see that **FLEET** is almost always better than other schemes for CI parameters between 7.5ms and ~50-70ms (except when bandwidth is too low). This is a very useful result since the Bluetooth SIG recommended BLE connection parameters are in this range. For example, iOS defaults to 30ms CI and Android defaults to 30ms min interval and 50ms max interval (CONNECTION_PRIORITY_BALANCED setting). We see that non-trivial speedup up to 4× can be achieved by **FLEET**.

Having described the overall results, we now look at each of the local and remote processing categories in more detail. The graphs in the rest of this section show one particular setting i.e. the Cortex-A77@200Mhz and BLE radio operating at 10ms CI, 8 Packets per CI at 1Mbps PHY running MobileNetV3 on ImageNet-100.

FLEET vs. Local Early Exit: We now look at how **FLEET** compares against state-of-art resource-optimized local early exit baselines — Shallow-Deep, BranchyNet and Ensemble.

Table 3 (Column "Local Early Exit") shows that **FLEET** works best for low to medium CIs, and its performance increases with more bandwidth and more packets per burst. This is intuitive since longer sleeps between transmissions would increase delay for communication, and more packets or bandwidth would reduce communication delay.

When the CI exceeds the regime shown in Table 3, local early-exit outperforms **FLEET**. When connection interval increases, the performance of **FLEET** is affected due to increasing offload delays whereas local early-exit is not affected. For example, when CI is 100ms, local early-exit achieves 1.5× speedup over **FLEET** on the Cortex-M33 (64 MHz) and the Cortex-A77 (200) MHz, and 3.3× speedup on more the more powerful GAP8 running at 175 MHz.

Figures 7 shows the latency comparison between our method and the local-early exit baselines. We see that **FLEET** achieves around 3× latency speed-up compared with baselines without sacrificing accuracy; the speedup can be even higher if we can tolerate some loss of accuracy.

This performance gap highlights one of the key advantages of offloading early exit computation to the cloud. Local early-exit directly uses features from the early stage of the deep learning model, pools them and uses them as input to a prediction head. However, since the features from the early stages tend to be coarse, local early-exit is unable to make sufficiently good decisions to exit early. However, **FLEET** *offloads early exit* computation to the cloud, hence it can use more resources to continue to extract fine-grained features from the early layers, which results in much better accuracy.

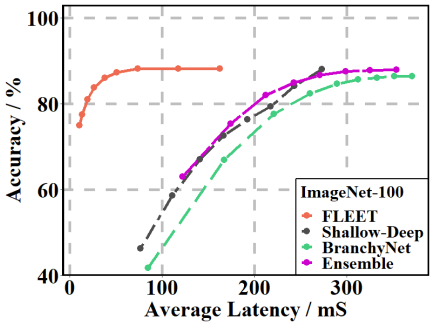


Figure 7: Latency of FLEET vs local early-exit.

Processors /#Pakcets /**PHY**	Local Early Exit		Model Partition		JPEG		All	
	CI / ms	Speed-Up	CI / ms	Speed-Up	CI /ms	Speed-Up	CI / ms	Speed-Up
Cortex-M33@64MHz	7.5 - 66	1 - 2.3x	18 - 64.5	1 - 1.2 x	18 - 100	1 - 1.7x	18 - 64.5	1 - 1.2x
Cortex-A77@200MHz	7.5 - 64.5	1 - 4.7x	7.5 - 4s	2.3 - 3.2x	7.5 - 4s	1.8 - 3.4x	7.5 - 64.5	1 - 3.1x
GAP8@175MHz	7.5 - 46.5	1 - 2.7x	7.5 - 4s	2.1 - 3.1x	7.5 - 4s	1.5 - 3.4x	7.5 - 46.5	1 - 2.6x
2	7.5 - 25.5	1 - 3.2x	7.5 - 4s	3.4 - 4.3x	7.5 - 4s	5.2 - 5.5x	7.5 - 75	1 - 3.2x
4	7.5 - 45	1 - 4.6x	7.5 - 4s	3.1 - 4.0x	7.5 - 4s	3.6 - 4.9x	7.5 - 45	1 - 4.0x
6	7.5 - 63	1 - 4.7x	7.5 - 4s	3.2 - 3.9x	7.5 - 4s	2.5 - 4.7x	7.5 - 63	1 - 3.9x
8	7.5 - 64.5	1 - 4.7x	7.5 - 4s	2.3 - 3.2x	7.5 - 4s	1.8 - 3.4x	7.5 - 64.5	1 - 3.1x
125 Kbps	-	-	7.5 - 4s	3.5 - 3.9x	7.5 - 4s	5.8 - 6.1x	-	-
500 Kbps	7.5 - 48	1 - 3.2x	7.5 - 4s	2.7 - 4.1x	7.5 - 4s	4.2 - 5.5x	7.5 - 48	1 - 3.2x
1Mbps	7.5 - 63	1 - 4.5x	7.5 - 4s	2.3 - 3.9x	7.5 - 4s	2.7 - 4.0x	7.5 - 63	1 - 3.9x
2Mbps	7.5 - 63	1 - 4.7x	7.5 - 4s	2.3 - 3.2x	7.5 - 4s	1.8 - 3.4x	7.5 - 63	1 - 3.1x

Table 3: Latency comparison of FLEET vs Local Early-Exit, Model Partition and JPEG baselines across different parameters. Column "CI" shows the range of connection intervals where FLEET achieves better performance than other methods i.e. has less latency. For brevity, each column reports the comparison between FLEET and *the best* among the baselines in each category. For local early exit, we compare FLEET against the best of the local methods i.e. BranchyNet, Shadow-Deep, and Ensemble (column 2). For Model Partition, we compare FLEET against the best of CLIO and NeuroSurgeon (column 3). The empty cells represent cases where FLEET does not achieve better performance than baselines. The Cortex-M33 is using CIFAR-100 images as input due to memory constraints and all other results are using ImageNet images. The comparisons between various schemes is for the same accuracy of 88% on ImageNet-100 and 68% on CIFAR-100.

FLEET vs. Model Partitioning: We now compare **FLEET** against several model partitioning-based baselines. In column "Model Partition" in Table 3, we see that **FLEET** is better than partitioned execution almost across the entire range of CI settings with up to 4.3× speedup (slightly narrower range for Cortex-M33). This is because while both **FLEET** and model partitioning methods are impacted by increasing CI, **FLEET** has pipelined computation and transmission, hence it achieves better overall latency in most cases.

The reason for the gap is simply because existing model partitioning methods execute in a sequential manner where they need to execute the models first and then transmit the features. In contrast, **FLEET** transmits in parallel with computation thereby exiting earlier than typical model partitioning schemes.

Figure 8 shows a more detailed comparison between **FLEET** and model partitioning baselines. We see that **FLEET** achieves less latency overall than the baselines for the same accuracy achieved. For example, we achieve around 5× reduction in latency to achieve 88% accuracy on the ImageNet-100 dataset.

FLEET vs. JPEG: We now compare **FLEET** against fully offloading data to the cloud for inference after compressing it using lossy JPEG compression. We use the different quality parameters in JPEG compression to generate the images of different sizes to fine-tune the model. Figure 9 shows a detailed comparison between **FLEET** and JPEG. We can see

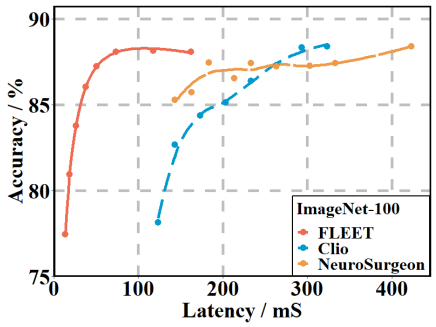


Figure 8: Latency of FLEET vs. model partitioning

that JPEG performs very poorly when it has to transmit at very low quality but it works better when higher quality data can be transmitted. In general, we see that **FLEET** achieves lower latency compared to JPEG.

Table 3 column "JPEG" shows that **FLEET** is almost always better than transmitting compressed data to the cloud. Only in one instance is this not the case i.e. for the Cortex-M33 at 64 MHz and CI at 7.5 ms, JPEG has 2.1× less latency compared with **FLEET**. In all other cases, **FLEET** is better. This is because compression + transmission is a sequential process and results in significant idle time on duty-cycled radios. In contrast, **FLEET** uses idle time to reduce the amount of data that needs to be transmitted. We find that **FLEET**

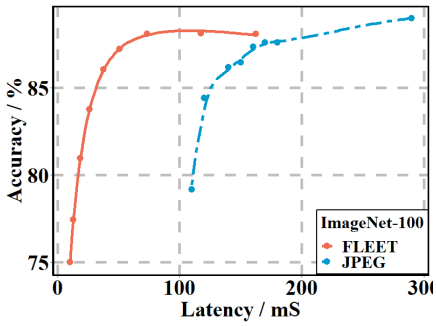


Figure 9: Latency of FLEET vs. jpeg

typically requires only a third of the data as JPEG-based data compression to achieves the same accuracy.

4.3 Impact of packet losses

So far, we have assumed no packet losses, but BLE has high packet losses, particularly when large packet sizes of more than 100 bytes are used and when transmission occurs in all CIs [41]. We evaluate the effect of packet loss with packet error rate from 0% to 90% in steps of 10%. **FLEET** with the dynamic pooling has the ability to downgrade to smaller feature sizes automatically.

We briefly describe how **FLEET** compares against other schemes as packet loss rate increases. For typical BLE settings of CI=25ms, 4pkts/CI and data rate of 500Kbps, **FLEET** can tolerate upto 18% packetloss before local early exit becomes the best approach in terms of latency. When we increase the number of packets to 8pkts/CI, **FLEET** is the best upto 40% error rate, and when the data rate is increased further to 1Mbps, **FLEET** tolerates upto 60% error rate. In all these cases, JPEG and partitioned execution perform worse than both **FLEET** and local early exit.

4.4 Impact of compute capability

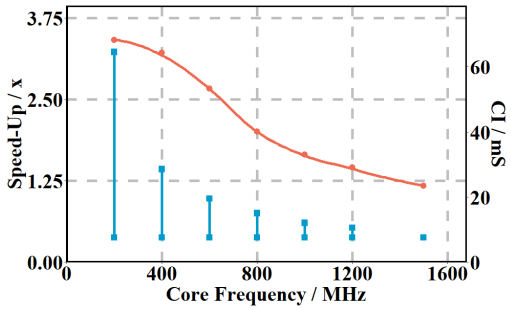


Figure 10: FLEET vs. the best of local early-exit, model split and JPEG baselines at different compute ability (Cortex-A77 frequency from 200 MHz to 1.5 GHz). The red curve is the latency speed-up and the blue segment is the CI range where FLEET achieves less latency.

So far, we looked at low to medium compute capability on IoT devices. Let us look at what happens as the compute capability increases beyond the range shown in Table 3.

To emulate platforms with different compute capabilities, we sweep through the frequency of Cortex-A77 from 200MHz to 1.5GHz with 100MHz spacing. For each setting, we measure the CI range where **FLEET** is better than alternatives and the maximum speedup obtained.

Figure 10 shows the results. We see that the benefits of **FLEET** is quite high for lower frequencies since local compute is slow and cloud offload is beneficial. The benefits diminish at higher frequencies since local processing becomes faster whereas the radio speed remains the same. A faster duty-cycled radio such as 802.11ah (WiFi HaLow) may be more appropriate for such higher frequency processors since it can offer higher data rates than BLE (up to 80Mbps when using 4 spatial channels).

4.5 Speedup for other models and datasets

So far, we have looked at MobileNetV3 and the ImageNet-100 dataset. We now look at performance of **FLEET** on more computationally heavy models and alternate datasets.

Performance on other models: We now look at the performance of **FLEET** using **ResNet34** and **InceptionV3** as the base model. We evaluate on the Cortex-A77@200Hz since these are larger models that require more resources.

Table 5 show the results. We see that **FLEET** can generalize to computationally heavy models and achieve substantial reduction in latency over local processing and cloud-offloading methods.

Performance for other datasets: We also evaluate **FLEET** on other datasets including **CIFAR-100** and **Tiny-ImageNet**. For CIFAR-100, we evaluate on the Cortex-M33 since the small image size is more appropriate for MCU-class platforms, and for Tiny-Imagenet, we evaluate on the Cortex-A77 since it is a medium-sized image.

Table 5 show the comparison between **FLEET**, local early-exit and model partition baselines using MobileNetV3. Due to the fact that the input size of CIFAR-100 is small(32x32x3), the speed-up of **FLEET** over baselines is not substantial since **FLEET** transmits small features after each layer. But on the Tiny-ImageNet dataset which has larger input size(64x64x3), **FLEET** can achieve higher latency speed-up 2.3x.

4.6 Energy Efficiency of FLEET

We now compare the energy-efficiency of various schemes. Energy is measured as the total cost of computation and communication. We calculate the energy consumption of computation in two ways: a) for the Raspberry Pi with Cortex-A77, we profile the device and empirically measure energy consumption, and b) for the Cortex-M33 and GAP8, we use

		PHY / Mbps	FLEET Energy / mJ	Baseline Energy / mJ	Efficiency
Local Early-Exit	Cortex-M33@64MHz	0.125 - 2	1.57 - 1.08	4.04	2.6 - 3.7x
	Cortex-A77@200MHz	0.125 - 2	1.41 - 8.11	35.81	2.5 - 4.4x
	GAP8@175MHz	0.5 - 2	1.83 - 1.04	2.26	1 - 2.1x
		PHY / Mbps	FLEET Energy / mJ	Baseline Energy / mJ	Efficiency
Model Partition	Cortex-M33@64MHz	0.125 - 1	1.57 - 1.08	4.15 - 1.25	1 - 2.6x
	Cortex-A77@200MHz	0.125 - 2	14.15 - 8.11	36.68 - 13.44	1.6 - 2.6x
	GAP8@175MHz	0.125 - 2	7.29 - 1.04	26.05 - 1.88	3.6 - 1.8x
		PHY / Mbps	FLEET Energy / mJ	Baseline Energy / mJ	Efficiency
JPEG	Cortex-M33@64MHz	0.125 - 0.5	1.57 - 1.08	7.01 - 1.25	1.1 - 4.4x
	Cortex-A77@200MHz	0.125 - 0.5	14.15 - 8.11	39.65 - 6.80	2.8 - 0.8x
	GAP8@175MHz	0.125 - 2	7.29 - 1.04	41.20 - 1.68	5.6 - 1.6x
		PHY / Mbps	FLEET Energy / mJ	Baseline Energy / mJ	Efficiency
All	Cortex-M33@64MHz	0.125 - 0.5	1.57 - 1.08	1.51 - 1.17	1 - 1.4x
	Cortex-A77@200MHz	0.125 - 0.5	14.15 - 8.11	35.81 - 6.80	2.5 - 0.8x
	GAP8@175MHz	0.5 - 2	1.83 - 1.04	2.26 - 1.68	1 - 1.6x

Table 4: The comparison of FLEET vs Local Early-Exit, Model Partition and JPEG baselines on the energy per inference across different processors. The Cortex-M33 is using CIFAR-100 image sizes due to memory constraints and all other results are using ImageNet image sizes.PHY means the range of physical layer throughputs in Mbps which FLEET is better and the Saving the range of energy saving that FLEET could achieve. In each category, FLEET is compared against the best baseline.

Models	All	
	CI / ms	Speed-up
ResNet34	7.5 - 52.5	1 - 4.5x
InceptionV3	7.5 - 49.5	1 - 4.2x
Datasets	All	
	CI / ms	Speed-up
CIFAR-100	18 - 64.5	1 - 1.2x
Tiny-ImageNet	7.5 - 19.5	1 - 2.3x

Table 5: FLEET vs Local Early-Exit, Model Partition and JPEG. The comparison across the different models is on ImageNet-100 and the comparison across the different datasets is using MobileNetV3. The CIFAR-100 results are on Cortex-M33@64MHz and the Tiny-ImageNet results are on Cortex-A77@MHz.

numbers provided by the datasheet to estimate energy consumption. The energy consumption for Bluetooth low energy is based on the highly detailed nRF Official Energy Profiler [4], which includes the energy cost of pre-processing, crystal ramp-up, radio start, radio TX, radio RX, switch, standby, and post-processing per CI. We use this to obtain the energy consumption of BLE under the different configurations. Among the BLE parameters, the physical layer bandwidth affects energy consumption the most since it changes the cost of using the radio (CI mainly affects latency), hence we compare for different PHY layer bandwidth settings.

Table 4 shows the results. We see that for lower end platforms like the Cortex-M33 on nRF5340DK, **FLEET** is as efficient as other schemes for low to medium bandwidth (between 125kbps and 750kbps). Since computational capability is low on this platform, compressed data transmission via JPEG performs best at higher bitrates. For faster processors such as the Cortex-A77, **FLEET** saves energy for low bandwidth. After about 500kbps, JPEG is more efficient. This is because compute power consumption is about 5× greater than the BLE radio power consumption, hence savings in data transmission size achieved by **FLEET** is not as high as the added energy cost of processing. Interestingly, we see energy benefits on the GAP8 even though the GAP8 is slightly more powerful than the Cortex-A77@200MHz. This is because the GAP8 is a neural accelerator and is much more efficient than the Cortex-A77 for DNNs (as shown in Table 2). So, we conclude that on low-end MCUs and optimized neural processors, **FLEET** can be expected to provide energy savings over alternate approaches.

4.7 Case study: Visual Wake Word

We implemented an end-to-end use case of **FLEET** inference pipeline for detecting visual wake words. This is a common use-case for low-power IoT platforms which wakeup upon

detecting whether a target of interest is present in the scene. If so, it triggers post-processing (e.g. security alarm) or interaction (e.g. smart doorbell). In this case study, we choose three sets of visual wake-up words from the ImageNet-100 dataset to mimic classes typically detected by lawn or backyard cameras — vehicles (ambulance, trucks, etc.), pets (different breeds of dogs and cats), and animals (snakes, coyotes etc.). We uses **ONNX** runtime to execute the IoT model on the Raspberry Pi running at 0.5 GHz and deploy the different clouds models on a laptop. We use **PyBluez** and **Gattlib** to transmit the features which after compression via **gzip**.

The overall accuracy of detecting visual wake-up is 93.96% and the average latency is 48.8ms, 1.6× less than local early-exit method which takes 78.6ms for the same accuracy. This is because for most of the data cases, **FLEET** early stops within the first **3** layers with cloud-assistance whereas local early-exit executes until layer 5.

5 RELATED WORK

Many different approaches have been proposed to enable execution of DNNs on low-power embedded platforms. We focus on three most relevant directions, partitioned execution, early exit, and model compression. None of these approaches have been designed to work with or evaluated on duty-cycled radios.

Partitioned Execution: Partitioned execution aims to partition a DNN so that local device processes only part of the model execution and the rest is offloaded to other devices or clouds. One common problem in partitioned execution is to find out the best partition point given a DNN model. [20, 40] profile the energy consumption and total latency of the model to find out the best point to partition the model. [27] leverages the quantization techniques to find the best partition point. [10] compares different ways in which IoT devices and cloud can cooperate including: local only, cloud only and combined and found out that the local or cloud only is not the best solution in terms of energy or latency.

Meanwhile, some works [5, 7, 17, 18] pay attention to the intermediate features when split the models across the edge and cloud. [5, 7] utilize the lossy compression techniques to compress the intermediate features. [17] enables the features robust to the network dynamics which can affect the performance of split models on the clouds. [18] leverages the dropout layers to fit the intermediate features into the packet loss scenarios.

Early Exit: Early exit reduces the compute cost of the model for less energy consumption and execution latency. [39] introduces the initial idea of early exit to make the deep learning models smaller. [43] mentioned the two different inference paradigm in early-exit methods: one is the input adaptive inference where only one early-exit and subnet will be chosen for the best energy or latency; the other is resources adaptive inference where the network will continue to generate the "anytime" predictions until the final output. [21] frames the early-exit as a solution to the overthinking problem of the deep learning models. The overthinking problem refers to that correct results are already achieved before the final layer of the model, which means a waste of latency and energy to continue the execution.

Model Compression: Other ways to deploy the deep learning models on resource-constrained devices include model compression, knowledge distillation, and specialized networks, all of which targets at a smaller compute and are usually orthogonal to the two aforementioned directions. Model pruning [11] and model quantization [19, 29] techniques are generic and can be applied to both the partitioned execution and early-exit methods. They either reduces the number of the neurons in the models or the number of the bits of the feature representation to achieve less compute. Knowledge Distillation [14, 28] leverages the features from the larger and deeper "teacher" models to train smaller and shallower "student" models by forcing the student models to learn and capture the distribution of the features from the teacher models. Some other works [26, 31] focus on training dynamic networks that can adapt to input and contexts during inference by executing different sub-networks.

6 CONCLUSION

In conclusion, our work in this paper introduces a new idea, cloud offload of early exit computation, to enable a powerful new paradigm for IoT-cloud inference that aligns with the duty-cycled operation of IoT radios. We address a number of technical challenges in designing such a cloud-offload based early-exit system (called **FLEET**) and show that this approach has substantial performance benefits by evaluating **FLEET** extensively across radio duty-cycling settings, different datasets, and multiple platforms. Our approach can be an important piece of the puzzle for deploying complex deep learning models on resource-constrained IoT platforms.

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 by National Science Foundation under Grant No. 1719386 and No. 1815347. 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.

REFERENCES

[1] Arm cortex-a77. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a77.

[2] Arm cortex-m33. https://developer.arm.com/Processors/Cortex-M33.

[3] Maximize ble throughput. https://punchthrough.com/maximizing-ble-throughput-on-ios-and-android/.

[4] nrf online power profiler. https://devzone.nordicsemi.com/power/w/opp.

[5] 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.

[6] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*, 2017.

[7] 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.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[9] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. A survey on ensemble learning. *Frontiers of Computer Science*, 14(2):241–258, 2020.

[10] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing*, 2019.

[11] 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.

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

[13] Himax WE-I Plus EVB Endpoint AI Development Board. https://www.sparkfun.com/products/17256.

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

[15] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[16] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B. Nightingale. Weardrive: Fast and energy-efficient storage for wearables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 613–625, Santa Clara, CA, July 2015. USENIX Association.

[17] 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.

[18] 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.

[19] 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.

[20] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[21] 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.

[22] Jong Hwan Ko, Taesik Na, Mohammad Faisal Amir, and Saibal Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018.

[23] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research).

[24] 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. IEEE, 2018.

[25] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.

[26] 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.

[27] Guangli Li, Lei Liu, Xueying Wang, Xiao Dong, Peng Zhao, and Xiaobing Feng. Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge. In *International Conference on Artificial Neural Networks*, pages 402–411. Springer, 2018.

[28] 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.

[29] 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.

[30] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Memory-efficient patch-based inference for tiny deep learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[31] Sicong Liu, Bin Guo, Ke Ma, Zhiwen Yu, and Junzhao Du. Adaspring: Context-adaptive and runtime-evolutionary deep model compression for mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(1):1–22, 2021.

[32] Zihao Liu, Tao Liu, Wujie Wen, Lei Jiang, Jie Xu, Yanzhi Wang, and Gang Quan. Deepn-jpeg: a deep neural network favorable jpeg-based image compression framework. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[33] 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.

[34] microTVM: TVM on bare-metal. https://tvm.apache.org/docs/topic/microtvm/index.html.

[35] Raspberry Pi. Raspberry pi 4 model b. *online].(https://www. raspberrypi. org*, 2015.

[36] S32R2X: Microcontrollers for High-Performance Radar. https://www.nxp.com/products/processors-and-microcontrollers/power-architecture/s32r-radar-mcus/s32r26-and-s32r27-microcontrollers-

for-high-performance-radar:S32R2X.

[37] Wenqi Shi, Yunzhong Hou, Sheng Zhou, Zhisheng Niu, Yang Zhang, and Lu Geng. Improving device-edge cooperative inference of deep learning via 2-step pruning. *arXiv preprint arXiv:1903.03472*, 2019.

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[39] 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.

[40] 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.

[41] Vishal Varun Tipparaju, Kyle R Mallires, Di Wang, Francis Tsow, and Xiaojun Xian. Mitigation of data packet loss in bluetooth low energy-based wearable healthcare ecosystem. *Biosensors*, 11(10):350, 2021.

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

[43] Yue Wang, Jianghao Shen, Ting-Kuei Hu, Pengfei Xu, Tan Nguyen, Richard Baraniuk, Zhangyang Wang, and Yingyan Lin. Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):623–633, 2020.

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