An Analysis of Network Overhead in Distributed TinyML

Ket Hollingsworth
Department of Computer Science
Harvey Mudd College
Claremont, CA
khollingsworth@g.hmc.edu

Sean Nian
Department of Computer Science
San Jose State University
San Jose, CA
seannians71@gmail.com

Alan Gutierrez
Department of Computer Science
Harvey Mudd College
Claremont, CA
alagutierrez@g.hmc.edu

Arthi Padmanabhan
Department of Computer Science
Harvey Mudd College
Claremont, CA
arpadmanabhan@g.hmc.edu

2024 IEEE/ACM Symposium on Edge Computing (SEC) | 979-8-3503-7828-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SEC62691.2024.00051

***Abstract*—This paper presents an implementation of a distributed TinyML system and an analysis of the network overhead in communication between devices. We use a pipelined approach, where each device computes a portion of the layers and sends intermediates to the next device. Our results show that the communication overhead strongly dominates, taking over five times longer than computation time. We present possible solutions to mitigate this overhead, including smart partitioning of models, optimizing device placement and signal strength, and alternate wireless protocols such as WiFi. These strategies aim to make the deployment of several TinyML devices working together a more practical scenario.**

I. Introduction

In recent years, there has been increased interest in running ML on edge devices rather than sending data to the cloud for ML processing. Major advantages of this approach include increased privacy [1], [2], reduced latency by avoiding sending high-density data, e.g., video, over the network [3], [4], and resilience to network disconnection [5]. As a result, much work explores how to effectively run ML on edge devices, which tend to be smaller and more resource-constrained [6], [7]. The field of TinyML takes this challenge to the extreme, exploring how to run ML on devices with extremely limited memory and milliwatt range power [8], [9], [10].

Machine learning models have been consistently getting larger to achieve higher accuracy, as seen by the increasing number of parameters in common models [11]. This trend makes the limitations of TinyML devices more apparent. A single TinyML device does not have the memory to run a large model; that is, it cannot fit the parameters of such models into its limited memory. Further, such low power devices quickly run out of energy trying to run complicated compute tasks such as ML.

Several techniques, such as pruning and quantization, can lower the size of the model such that it fits on a single TinyML device. However, such techniques can lower the accuracy of the model [12]. This makes a single TinyML device infeasible for many realistic applications, as deployments tend to have accuracy targets [13], [4] that they must meet.

Rather than using a single device, we consider the idea of using TinyML devices for swarm intelligence, where several devices work together on a single ML inference. Deploying several devices together is a common strategy, given the modest price point and high rate of failure of low power devices [14], [15]. Several devices working together offers a promising solution to the accuracy problem by enabling larger models to run.

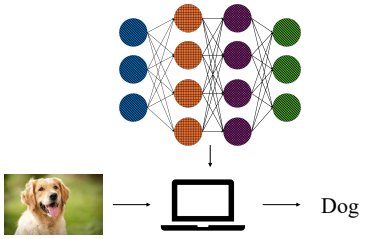
We explore the technique of pipelining inference [16], [17]. That is, each device takes a portion of the model's layers, as seen in Figure 1. While each device cannot individually fit the whole model, the two devices together can, as Device 1 runs the first two layers, sends the intermediate values to Device 2, and Device 2 completes the inference by running the last two layers. This strategy eases the memory constraint on a single device and allows the devices to share the computational load. Additionally, this strategy offers the opportunity to increase throughput by pipeline parallelization, where Device 1 could start running inference on the subsequent data sample while Device 2 is still finishing the first sample.

A major challenge in setting up a usable distributed ML inference is the overhead associated with sending information between devices. Several prior works on ML across resource-constrained devices mention this overhead or simulate network conditions [18], [19], [17], [20], [21]. However, current literature is missing a deep-dive into this overhead and what comprises it.

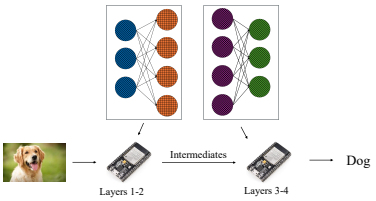
This preliminary work presents a case study of running MobileNet as a pipelined inference across 5 ESP32-S3 microcontrollers. We quantify the latency due to network in our system and compare it with the latency due to inference. Then, we analyze the factors that contribute to this overhead. Finally, we propose promising directions to address the overhead and take steps towards a realistic distributed TinyML deployment.

This work is supported in part by NSF grant CNS-2243941.

2837-4827/24/$31.00 ©2024 IEEE
DOI 10.1109/SEC62691.2024.00051



(a) Workflow when the full model fits on a device



(b) Workflow when model is split into multiple devices

Fig. 1: Splitting a model onto two devices

II. System Setup

We describe the setup and design choices of our system, which splits a model among several TinyML devices, sending the intermediates between devices.

A. Devices

For our devices, we used the ESP32-S3, a common choice in TinyML applications [22], [23], [24], [25]. The ESP32-S3 is a low power microcontroller developed by Espressif with 512KB SRAM, 4Mb flash, and a clock speed of 240MHz. It also provides a partition table that allows users to specify how the device's flash memory is organized and allocated (allowing us to fit more code and weights on each device), native support for wireless communication protocols like WiFi and Bluetooth, and low-power sleep modes that significantly reduce power consumption. We chose the ESP32-S3 because of its support for communication protocols as well as the significant support from both Espressif and the ESP32 user community for running ML models. The ESP-DL library, developed by Espressif, provides instructions and examples for running small ML models on the ESP32-S3.

B. Models

We started with a MobileNet classification model in Tensorflow and modified it to enable it to run on an ESP32-S3, as the ESP-DL library only supports a specific format and certain layer types. We replaced ReLU6 layers, which are unsupported, with ReLU layers. We then converted the model to work for 10 classes, matching our intended training set, Imagenette [26], which is a subset of Imagenet with only 10 classes. To do this, we added a Reshape layer to change the tensor shape, a Conv2D layer to apply convolution with 10 classes, and a Flatten layer to convert the data into a one-dimensional format before the dense layer. After modifying and training the model, we used the tools provided in the ESP-DL toolkit to convert the TensorFlow model into an ONNX model, the only format supported to run on the ESP32-S3. After conversion, we quantized the model into an int16 model to make it compatible with the ESP-DL library and deployable on the ESP32-S3. We note that while we aimed at running a larger model as opposed to exclusively using quantization to make it smaller, we did the minimum quantization possible to be able to run the model on the ESP32. Our final model had 67 layers total (36 convolutional), an accuracy of 70%, and was 8.9MB in memory.

C. Model Splitting

When splitting models, we considered all components that have to fit on a model for it to receive intermediates, run an inference, and send the next set of intermediates. This includes 1) the parameters of all model layers assigned to the device 2) the code for both running and sending using Bluetooth Low Energy, discussed below, and 3) the maximum of all intermediates for the set of layers on this device. The memory related to intermediates includes the intermediates received from the previous device (or the size of the input if this is the first device in the pipeline) as well as the final set of intermediates generated from this device (or the final result if this is the last device in the pipeline). Note that we consider the maximum of the intermediates because they do not all need to be held in memory at the same time.

Based on the size of our model (8.9Mb), the memory on each ESP32-S3 (∼4Mb), the size of the code (∼0.4Mb), and the size of the intermediates, we used five devices. We initially took a greedy approach, fitting as many layers as possible on device 1, then repeating the process for device 2, etc. We then manually edited the placement of certain layers, without changing the number of devices, to lower the size of the intermediates. The division of layers and memory can be seen in Table I and the five devices can be seen in Figure 2.

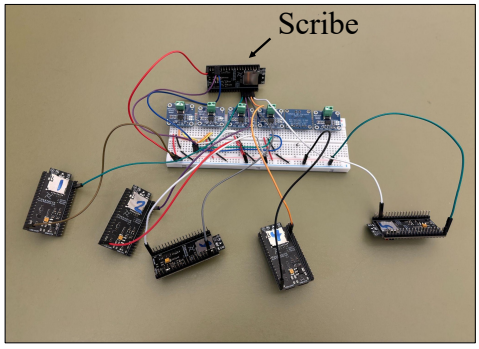


Fig. 2: System setup with 5 ESP32-S3 devices running MobileNet and sending with BLE

D. Network Communication

Our system uses Bluetooth Low Energy (BLE) because it is ideal for applications where energy efficiency is crucial,

ESP32-S3 Device	Layers Assigned	Memory (MB)
Device 1	1-23	1.13
Device 2	24-37	1.37
Device 3	38-49	1.69
Device 4	50-59	2.73
Device 5	60-67	2.42

TABLE I: Distribution of model layers across ESP32-S3 devices

consuming about 100x less power than classic Bluetooth [27]. This is because instead of being always on, BLE stays in sleep mode unless it has an active connection. BLE allows for direct device-to-device communication with a range of 70-100m and operates independently of major service providers. On the ESP32-S3, BLE supports a maximum packet size of 517 bytes.

Since BLE uses its own protocol stack rather than the traditional OSI, our code had to add functionality, such as breaking data into chunks to fit in BLE packets. In our system, each packet included a 3-byte overhead imposed by BLE, an 8-byte timer ID for data measuring purposes, and the intermediates of the inference in 252 16-bit integers.

Because of the lack of built-in TCP functionality, our code also had to choose when to require an acknowledgment when sending several packets. In some systems that use BLE, such as Android, the device has a mechanism for flow control. Such devices will automatically handle full buffers and only write to the buffers when there is space. Others, like iOS and ESP32, do not have such a mechanism and will instead silently drop packets when buffers are full. Therefore, we had to effectively choose a window size n, implemented by requiring a response for every n^{th} packet.

We determined the ideal window size by testing different window sizes across our pipeline of devices and finding the largest possible size that avoided dropped packets across 10 runs. The difference between devices is due to the different sizes of intermediates sent between them. We can see from Figure 3 that with higher window sizes, each device is more likely to fail, increasing the probability that some device in the pipeline fails. Devices 4 and 5 failures are not shown because if packets were going to get dropped, they did so within the first 3 of our 5 devices. We found that using a window size of 5 and 6 was similar, though 6 was marginally faster between certain devices, as shown in Figure 4. Therefore, we chose a window size of 6.

E. Measuring Sending Time

To measure the time to send intermediates from one device to another, we used a separate ESP32-S3 as a scribe, as shown in Figure 2. The scribe was connected to each device in the pipeline using a separate wire. For each device, when it started sending, it changed the voltage on the wire connecting it to the scribe from high to low, prompting the scribe to be interrupted and immediately record its current time. When a device finished receiving data, it changed the voltage on its wire connecting it to the scribe from low to high, similarly prompting the scribe to record the time. In this way, the scribe kept a log of all times to send data through the pipeline. Inference times were calculated by taking the time in between when a device received from the previous device and when it sent to the next device, and these times were validated against each device's own recording of its inference time.

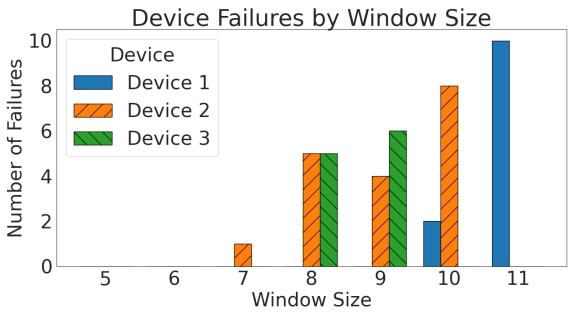


Fig. 3: Failure rate of each device as window size is changed

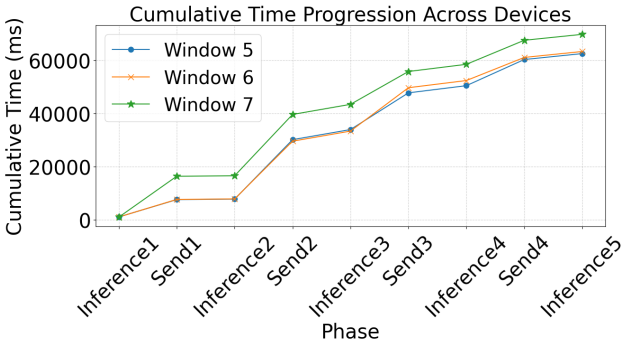
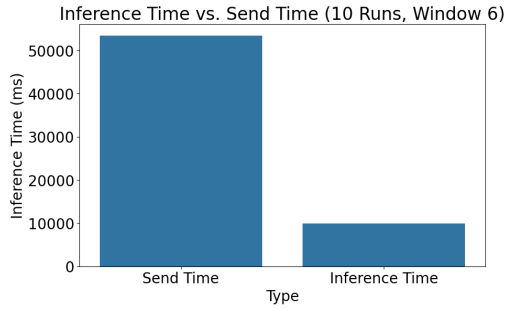


Fig. 4: Cumulative Time to send across pipeline as window size is changed

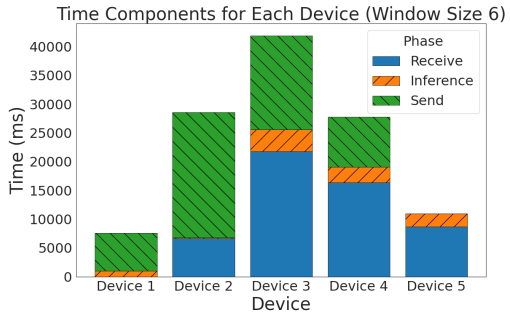
III. Network Overhead Quantification

Latency. Here, we quantify and assess the source of the network overhead in terms of latency. We can see from Figure 5a that the latency from sending data between devices outweighs the latency from performing inference on the devices by over five times. Figure 5b shows the proportions of time for sending, receiving, and inference, broken up by device. We compare the two main factors that contribute to this: the number of packets sent and the signal strength between devices.

Number of packets. We look at how the number of packets, which determines the size of the intermediates, affects latency of sending and receiving. It is intuitive that sending more data would take more time, but the effect is amplified by the memory constraints of tiny devices. This is because we must set the window size such that packets are not dropped when the device's buffers are filled, as described in Section 2. With the ESP32-S3's memory constraints, its buffers fill up quickly, forcing us to choose a small window size and therefore adding delay for the sake of not dropping packets. Figure 6 shows the



(a) Time to Send over the Network Compared with Time to Run Inference



(b) Per Device Time to Send, Receive, and Run Inference

Fig. 5: Breakdown of Time Spend on Each Phase of Pipelined Inference

number of bytes in the intermediates (corresponding to the number of packets, as all packets were maximally filled) and the time for all intermediates to be received. While we can see a moderate correlation here (correlation coefficient = 0.56), we can also look at signal strength to explain discrepancies.

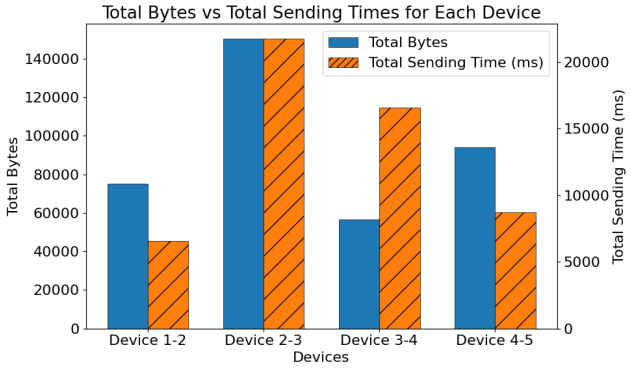


Fig. 6: Time to Send Between Devices and Total Bytes Sent

Signal Strength. We expect that devices with a better signal strength (affected by both distance and interference) will be able to send data between each other faster. Here, we analyze that effect. We measured the signal strength of the four connections between the five devices in our pipeline, and these are shown in Figure 7 along with the time for all intermediates to be received. We can also observe correlations here, as device pairs with a stronger signal (closer to 0) have lower latencies. In fact, this correlation is stronger than that for the number of packets (correlation coefficient = 0.99), indicating that adjusting the position of devices to improve signal strength is the most promising method to lowering network latency.

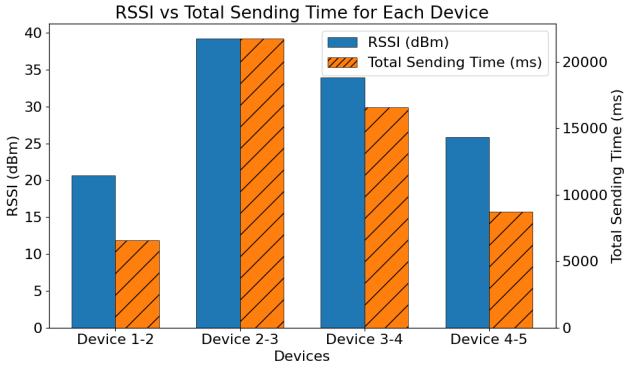


Fig. 7: Time to Send Between Devices and Signal Strength

IV. Promising Directions

Using Signal Strength to Optimize for Pipelined Inference. Since we identified signal strength to be the biggest contributing factor to latency, we explore how to use this information to optimize pipeline inference. Recall from Section 1 that in pipelined inference, a device can start running inference on the next sample as soon as it sends its portion of work to the next device. In pipelined inference, if there is a large discrepancy between the times that each device takes to receive, process, and send its portion, that device becomes the bottleneck in the pipeline. To avoid this, we aim to make the devices' end-to-end times as even as possible.

We saw from Figure 5b that some devices take longer to send their data than others. We validated that the signal strength correlation we saw in our system generalizes by varying the distance between two devices and measuring signal strength and RTT, shown in Figure 8. We believe that the system could be optimized by careful placement of devices to offset the differences in intermediate size. An example is shown in Figure 9 below. Since B sends a larger set of intermediates to C than A does to B, B and C are positioned closer together. This way, we can optimize for the A-B and B-C communication times to be similar such that pipelined inference can run without either link being a bottleneck.

Smart Partitioning. Since we have identified that the number of packets in the intermediates is a contributing factor to latency, this direction is aimed at lowering the size of the intermediates. Smart partitioning involves dividing a model at pooling layers to minimize the amount of data sent between devices. Pooling layers, such as max pooling and average pooling, are used to reduce the spatial dimensions of feature maps in neural networks. They achieve this by summarizing regions of the input, either by taking the maximum value

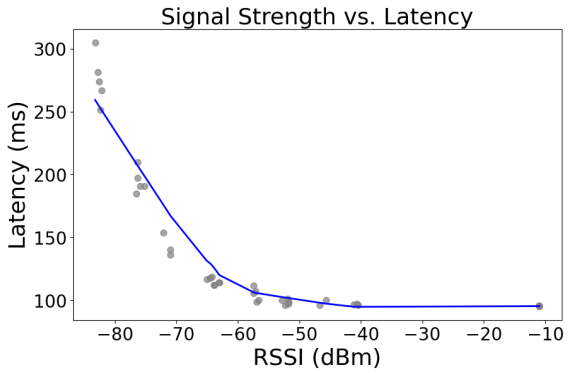


Fig. 8: Latency vs RSSI values

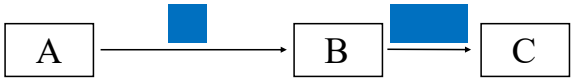


Fig. 9: Sample Positioning of Devices

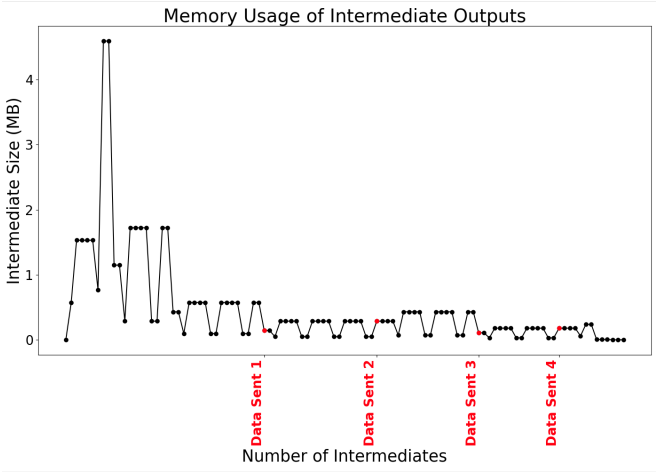


Fig. 10: Sizes of intermediates in our model. Those shown in red are sent across devices.

(max pooling) or the average value (average pooling), which retains the most significant features and discards redundant information [28]. Pooling layers make a model more efficient and easier to process by reducing the amount of data the model needs to handle at each subsequent layer. Therefore, strategically splitting models at pooling layers or any other layers that reduce data minimizes that the amount of information transferred between devices, leading to faster processing times and lower communication costs [29].

In Figure 10, we can see the sizes of each set of intermediates in our model, with the intermediates sent between devices shown in red. As described in Section 2, we initially used a greedy approach and then manually moved layers where possible to lower the size of the intermediates sent between devices. As this process was not trivial and did not guarantee the lowest possible intermediates sent, we believe our system could be optimized by automating the process of choosing which layers belong to which device while minimizing both the number of devices and the size of the intermediates sent between them.

Smart Replication. We consider how we might scale this system out if we had a swarm of devices, rather than just the 5 needed to run our system. For example, if we had 20 devices, we could add redundancy by keeping 4 copies of each device's portion. However, since each device takes a different amount of time to run and send, we could consider a smarter form of replication, where we replicate the devices that are likely to be bottlenecks. Similar to optimizing geographical position, this could increase the throughput in pipelined inference. While this work focuses more on latency, we could also see smart replication being useful for balancing energy consumption. If some devices consistently use more energy, they will die faster, so we can increase the longevity of the system by replicating those devices more.

Consider Other Communication Protocols.

We chose to use Bluetooth Low Energy because ESP32-S3s are extremely low power devices. BLE is known for being energy efficient, but it is also recommended to be used for sending small amounts of data. The high latency of communication in our system, combined with our findings of how the number of packets affects latency, makes us consider whether other communication protocols, such as WiFi, might be a better fit for distributed ML.

While WiFi is not particularly energy efficient, its high data transfer speed offers an alternative to the low speeds of BLE, as seen in Table II. In future work, we can consider whether the higher energy cost of WiFi is worth the high speed it provides. We would also consider, though, that WiFi must always be on, as opposed to BLE which can sleep and use wake-up signals, leading to energy consumption with WiFi even when not actively sending and receiving.

	BLE	**WiFi**
Data Throughput	90Kb/s	2500Kb/s
Maximum Current	2.4mA - 130mA	130mA - 240mAh
Sleep Enabled	Yes	No

TABLE II: Comparison of BLE and WiFi

A promising direction is using WiFi for data transfers in conjunction with BLE for wake-up signals, which could provide a faster, energy-efficient system. However, we must also consider that many TinyML use cases involve remote scenarios without internet connection, and if WiFi were used without connection to the internet, one device would have to act as an access point, resulting in a single point of failure.

Device Specific Window Size Choice

Our system uses a fixed window size across all device pairs. We chose this by finding the largest window size such that across the pipeline, packets were not dropped due to filled

buffers. However, while finding this window size, we noted that the optimal window size appears different for different device pairs, perhaps due to signal strength or slight chip differences. We did not explore finding the optimal window size per pair, but future work could consider automating the process of dynamically choosing the window size.

V. Related Work

Prior work has targeted several methods to effectively run ML on low power devices. Federated learning has been explored by [20], which combines federated learning and transfer learning to train models on IoT devices while preserving data privacy collaboratively. They note that network discrepancies between devices cause large differences in sending time and they therefore develop an algorithm to allow faster nodes to proceed without waiting for slower ones. [30] develops an algorithm to collaboratively learn an NN initialization across several tiny devices. They similarly note the latency differences between devices due to communication overheads and address this using a serial communication schema. TinyFedTL [21] presents an implementation of Federated Learning using MobileNet on the Arduino Nano 33. TinyFedTL similarly notes that sending time outweighed the time to perform the ML.

Load Balancing and optimizations for edge devices are active research topics [31][32][33]. Load balancing in [31] studies deadline-aware task scheduling and dispatching with bandwidth constraints using optimization of networking and computing resources. [32] proposes a strategy called semi-dynamic load balancing to address the issue of stragglers in distributed ML workloads. [33] explores different caching algorithms for edge devices aimed at solving cold start problems by maximizing the cache hit rate and minimizing cold start delays.

Frameworks for effective model partitioning have been explored in [17][34][35]. [34] explores dividing a CNN structure into a set of partitions whose size is determined by the constrained resources of the edge devices. Works like [17][35] present an algorithm that partitions DNNs and distributes them across a set of edge devices with the goal of minimizing the bottleneck latency and, therefore, maximizing inference throughput.

Finally, other approaches [36] use an alternative to a pipelined approach, such as tiling, or finding portions of the network that can be computed independently. This can be used to run distributed inference [29] or to offload some portions of the network [37].

VI. Conclusion

To make distributed TinyML systems more efficient for real-world applications, we must understand and mitigate the large overhead caused by communication between such low-power, low-memory devices. With this workshop paper, we take a step towards better understanding this overhead by implementing a distributed scenario, pipelined inference. Based on our analysis of the bottlenecks, we map future directions that can address the core issues in attempting to deploy this scenario.

References

[1] Microsoft. enabling data residency and data protection in microsoft azure regions. https://azure.microsoft.com/en-us/resources/achieving-compliant-data-residency-and-security-with-azure/), 2021. Accessed: 2024-08-15.

[2] Data act: Shaping europe's digital future. https://digital-strategy.ec.europa.eu/en/policies/data-act). Accessed: 2024-08-15.

[3] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 129–143, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Ofcom. residential landline and fixed broadband services. https://www.ofcom.org.uk/__data/assets/pdf_file/0015/113640/landline-broadband.pdf), 2017. Accessed: 2024-08-01.

[6] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Comput. Surv.*, 54(8), oct 2021.

[7] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Video analytics - killer app for edge computing. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 695–696, New York, NY, USA, 2019. Association for Computing Machinery.

[8] tinyml foundation. https://www.tinyml.org/). Accessed: 2024-08-15.

[9] Partha Pratim Ray. A review on tinyml: State-of-the-art and prospects. *Journal of King Saud University - Computer and Information Sciences*, 34(4):1595–1623, 2022.

[10] Colby Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, D. Pau, Jae-sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. Benchmarking tinyml systems: Challenges and direction, 03 2020.

[11] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, An Chang Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *ArXiv*, abs/2207.02852, 2022.

[12] Riku Immonen and Timo Hämäläinen. Tiny machine learning for resource-constrained microcontrollers. 2022.

[13] 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.

[14] Wenjing Li, Wei Deng, Rui She, Ningchi Zhang, Yanru Wang, and Wenjie Ma. Edge computing offloading strategy based on particle swarm algorithm for power internet of things. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 145–150. IEEE, 2021.

[15] P Kalyanakumar, S Srinivasa Pandian, S Boopalan, D Kani Jesintha, R Santhana Krishnan, and A Essaki Muthu. Harnessing bio-inspired optimization and swarm intelligence for energy-aware tinyml in iot. In *2024 International Conference on Inventive Computation Technologies (ICICT)*, pages 1226–1233. IEEE, 2024.

[16] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: Ml inference pipeline composition framework. *arXiv preprint arXiv:1812.01776*, 2018.

[17] Arjun Parthasarathy and Bhaskar Krishnamachari. Defer: Distributed edge inference for deep neural networks. In *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 749–753, 2022.

[18] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. The synergy of complex event processing and tiny machine learning in industrial iot. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, DEBS '21. ACM, June 2021.
[19] Nil Llisterri Giménez, Marc Monfort Grau, Roger Pueyo Centelles, and Felix Freitag. On-device training of machine learning models on microcontrollers with federated learning. *Electronics*, 11(4), 2022.
[20] M. Ficco, A. Guerriero, E. Milite, F. Palmieri, R. Pietrantuono, and S. Russo. Federated learning for iot devices: Enhancing tinyml with on-board training. *Information Fusion*, 104:102189, 2024.
[21] Kavya Kopparapu and Eric Lin. Tinyfedtl: Federated transfer learning on tiny devices, 2021.
[22] Kristian Dokic. Microcontrollers on the edge – is esp32 with camera ready for machine learning? In Abderrahim El Moataz, Driss Mammass, Alamin Mansouri, and Fathallah Nouboud, editors, *Image and Signal Processing*, pages 213–220, Cham, 2020. Springer International Publishing.
[23] Md Ziaul Haque Zim. Tinyml: analysis of xtensa lx6 microprocessor for neural network applications by esp32 soc. *arXiv preprint arXiv:2106.10652*, 2021.
[24] Chua Kiang Hong, Mohd Azlan Abu, Mohd Ibrahim Shapiai, Mohamad Fadzli Haniff, Radhir Sham Mohamad, and Aminudin Abu. Analysis of wind speed prediction using artificial neural network and multiple linear regression model using tinyml on esp32. *Journal of Advanced Research in Fluid Mechanics and Thermal Sciences*, 107(1):29–44, 2023.
[25] Parth Mahajan, Samarth Otari, Pratik Meshram, and Krishna Mhaske. Autonomous weed cutter leveraging esp32 and tiny ml. *Grenze International Journal of Engineering & Technology (GIJET)*, 10, 2024.
[26] Jeremy Howard. Imagenette: A smaller subset of 10 easily classified classes from imagenet. https://github.com/fastai/imagenette, 2019.
[27] Argenox. Bluetooth low energy (ble) faq, 2024.
[28] Keras Documentation. Pooling layers. https://keras.io/api/layers/pooling_layers/.
[29] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.
[30] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. Tinyreptile: Tinyml with federated meta-learning, 2023.
[31] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20. ACM, October 2020.
[32] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 2287–2295, April 2019.
[33] Austin Chen and Genya Ishigaki. Scaling container caching to larger networks with multi-agent reinforcement learning. In *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–5, July 2024.
[34] Nihel Kaboubi, Loïc Letondeur, Thierry Coupaye, Fréderic Desprez, and Denis Trystram. Hybrid partitioning for embedded and distributed cnns inference on edge devices. In Isaac Woungang, Sanjay Kumar Dhurandher, Kiran Kumar Pattanaik, Anshul Verma, and Pradeepika Verma, editors, *Advanced Network Technologies and Intelligent Computing*, pages 164–187, Cham, 2023. Springer Nature Switzerland.
[35] Arjun Parthasarathy and Bhaskar Krishnamachari. Partitioning and placement of deep neural networks on distributed edge devices to maximize inference throughput. In *2022 32nd International Telecommunication Networks and Applications Conference (ITNAC)*, pages 239–246, Nov 2022.
[36] Rafael Stahl, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Fused depthwise tiling for memory optimization in tinyml deep neural network inference. *arXiv preprint arXiv:2303.17878*, 2023.
[37] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.