SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud

Stefanos Laskaridis[†]*, Stylianos I. Venieris[†]*,
Mario Almeida[†]*, Ilias Leontiadis[†]*, Nicholas D. Lane[†,‡]
[†]Samsung AI Center, Cambridge [‡]University of Cambridge
* *Indicates equal contribution.*
{stefanos.l,s.venieris,mario.a,i.leontiadis,nic.lane}@samsung.com

ABSTRACT

Despite the soaring use of convolutional neural networks (CNNs) in mobile applications, uniformly sustaining high-performance inference on mobile has been elusive due to the excessive computational demands of modern CNNs and the increasing diversity of deployed devices. A popular alternative comprises offloading CNN processing to powerful cloud-based servers. Nevertheless, by relying on the cloud to produce outputs, emerging mission-critical and high-mobility applications, such as drone obstacle avoidance or interactive applications, can suffer from the dynamic connectivity conditions and the uncertain availability of the cloud. In this paper, we propose SPINN, a distributed inference system that employs synergistic device-cloud computation together with a progressive inference method to deliver fast and robust CNN inference across diverse settings. The proposed system introduces a novel scheduler that co-optimises the early-exit policy and the CNN splitting at run time, in order to adapt to dynamic conditions and meet user-defined service-level requirements. Quantitative evaluation illustrates that SPINN outperforms its state-of-the-art collaborative inference counterparts by up to 2× in achieved throughput under varying network conditions, reduces the server cost by up to 6.8× and improves accuracy by 20.7% under latency constraints, while providing robust operation under uncertain connectivity conditions and significant energy savings compared to cloud-centric execution.

CCS CONCEPTS

• **Computing methodologies** → *Distributed computing methodologies*; • **Human-centered computing** → *Ubiquitous and mobile computing*.

ACM Reference Format:
Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20), September 21–25, 2020, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372224.3419194

MobiCom '20, September 21–25, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7085-1/20/09…$15.00
https://doi.org/10.1145/3372224.3419194

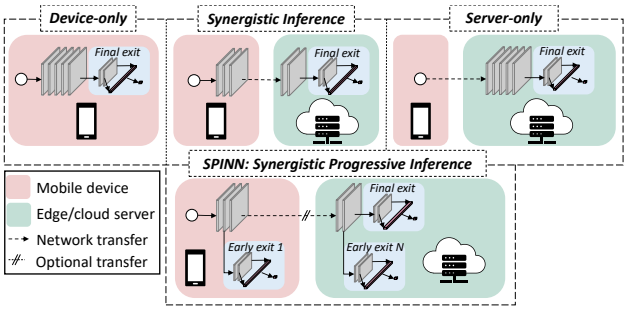

Figure 1: Existing methods vs. SPINN.

1 INTRODUCTION

With the spectrum of CNN-driven applications expanding rapidly, their deployment across mobile platforms poses significant challenges. Modern CNNs [20, 68] have excessive computational demands that hinder their wide adoption in resource-constrained mobile devices. Furthermore, emerging user-facing [80] and mission-critical [37, 42, 66] CNN applications require low-latency processing to ensure high quality of experience (QoE) [4] and safety [11].

Given the recent trend of integrating powerful System-on-Chips (SoCs) in consumer devices [2, 25, 78], direct on-device CNN execution is becoming possible (Figure 1 - top left). Nevertheless, while flagship devices can support the performance requirements of CNN workloads, the current landscape is still very diverse, including previous-gen and low-end models [80]. In this context, the less powerful low-tier devices struggle to consistently meet the application-level performance needs [2].

As an alternative, service providers typically employ cloud-centric solutions (Figure 1 - top right). With this setup, inputs collected by mobile devices are transmitted to a remote server to perform CNN inference using powerful accelerators [3, 6, 12, 19, 31, 32]. However, this extra computation capability comes at a price. First, cloud execution is highly dependent on the dynamic network conditions, with performance dropping radically when the communication channel is degraded. Second, hosting resources capable of accelerating machine learning tasks comes at a significant cost [40]. Moreover, while public cloud providers offer elastic cost scaling, there are also privacy and security concerns [64].

To address these limitations, a recent line of work [22, 34, 46] has proposed the collaboration between device and cloud for CNN inference (Figure 1 - top center). Such schemes typically treat the CNN as a computation graph and partition it between device and cloud. At run time, the client executes the first part of the model and transmits

the intermediate results to a remote server. The server continues the model execution and returns the final result back to the device. Overall, this approach allows tuning the fraction of the CNN that will be executed on each platform based on their capabilities.

Despite their advantages, existing device-cloud collaborative inference solutions suffer from a set of limitations. First, similar to cloud execution, the QoE is greatly affected by the network conditions as execution can fail catastrophically when the link is severely deteriorated. This lack of network fault tolerance also prevents the use of more cost-efficient cloud solutions, *e.g.* using ephemeral spare cloud resources at a fraction of the price.[1] Furthermore, CNNs are increasingly deployed in applications with stringent demands across multiple dimensions (*e.g.* target latency, throughput and accuracy, or device and cloud costs).[2] Existing collaborative methods cannot sufficiently meet these requirements.

To this end, we present SPINN, a distributed system that enables robust CNN inference in highly dynamic environments, while meeting multi-objective application-level requirements (SLAs). This is accomplished through a novel scheduler that takes advantage of progressive inference; a mechanism that allows the system to exit early at different parts of the CNN during inference, based on the input complexity (Figure 1 - bottom). The scheduler optimises the overall execution by jointly tuning both the split point selection and the early-exit policy at run time to sustain high performance and meet the application SLAs under fluctuating resources (*e.g.* network speed, device/server load). The guarantee of a local early exit renders server availability non-critical and enables robust operation even under uncertain connectivity. Overall, this work makes the following key contributions:

- A progressive inference mechanism that enables the fast and reliable execution of CNN inference across device and cloud. Concretely, on top of existing early-exit designs, we propose an early-exit-aware cancellation mechanism that allows the interruption of the (local/remote) inference when having a confident early prediction, thus minimising redundant computation and transfers during inference. Simultaneously, reflecting on the uncertain connectivity of mobile devices we design an early-exit scheme with robust execution in mind, even under severe connectivity disruption or cloud unavailability. By carefully placing the early exits in the backbone network and allowing for graceful fallback to locally available results, we guarantee the responsiveness and reliability of the system and overcome limitations of existing offloading systems.
- A CNN-specific packing mechanism that exploits the reduced-precision resilience and sparsity of CNN workloads to minimise transfer overhead. Our communication optimiser combines a lossless and an accuracy-aware lossy compression component which exposes previously unattainable designs for collaborative inference, while not sacrificing the end accuracy of the system.
- An SLA- and condition-aware scheduler that co-optimises i) the early-exit policy of progressive CNNs and ii) their partitioning between device and cloud at run time. The proposed scheduler employs a multi-objective framework to capture the user-defined importance of multiple performance metrics and translate them into SLAs. Moreover, by surveilling the volatile network conditions and resources load at run time, the scheduler dynamically selects the configuration that yields the highest performance by taking into account contextual runtime information and feedback from previous executions.

[1] AWS Spot Instances – https://aws.amazon.com/ec2/spot/.
[2] Typically expressed as service-level agreements (SLAs).

2 BACKGROUND AND RELATED WORK

To optimise the execution of CNN workloads, several solutions have been proposed, from compiler [1, 30, 65] and runtime optimisations [36, 43, 49] to custom cloud [7, 19, 34] and accelerator designs [75, 79]. While these works target a single model with device- or cloud-only execution, the increased computational capabilities of client devices [2, 25] have led to schemes that maximise performance via device-cloud synergy. Next, we outline significant work in this direction and visit approximate computing alternatives which exploit accuracy-latency trade-offs during inference.

Approximate Inference. In applications that can tolerate some accuracy drop, a line of work [9, 18, 45] exploits the accuracy-latency trade-off through various techniques. In particular, NestDNN [9] employs a multi-capacity model that incorporates multiple descendant (*i.e.* pruned) models to expose an accuracy-complexity trade-off mechanism. However, such models cannot be natively split between device and cloud. On the other hand, model selection systems [18, 45] employ multiple variants of a single model (*e.g.* quantised, pruned) with different accuracy-latency trade-offs. At run time, they choose the most appropriate variant based on the application requirements and determine where it will be executed. Similarly, classifier cascades [21, 33, 38, 39, 71] require multiple models to obtain performance gains. Despite the advantages of both, using multiple models adds substantial overhead in terms of maintenance, training and deployment.

Progressive Inference Networks. A growing body of work from both the research [23, 35, 72, 81, 84] and industry communities [55, 74] has proposed transforming a given model into a progressive inference network by introducing intermediate exits throughout its depth. By exploiting the different complexity of incoming samples, easier examples can early-exit and save on further computations. So far, existing works have mainly explored the hand-crafted design of early-exit architectures (MSDNet [23], SCAN [84]), the platform- and SLA-agnostic derivation of early-exit networks from generic models (BranchyNet [72], SDN [35]) or the hardware-aware deployment of such networks (HAPI [44]). Despite the recent progress, these techniques have not capitalised upon the unique potential of such models to yield high mobile performance through distributed execution and app-tailored early-exiting. In this context, SPINN is the first progressive inference approach equipped with a principled method of selectively splitting execution between device and server, while also tuning the early-exit policy, enabling high performance across dynamic settings.

Device-Cloud Synergy for CNN Inference. To achieve efficient CNN processing, several works have explored collaborative computation over device, edge and cloud. One of the most prominent pieces of work, Neurosurgeon [34], partitions the CNN between a device-mapped *head* and a cloud-mapped *tail* and selects a single split point based on the device and cloud load as well as the network conditions. Similarly, DADS [22] tackles CNN offloading, but from a scheduler-centric standpoint, with the aim to yield the

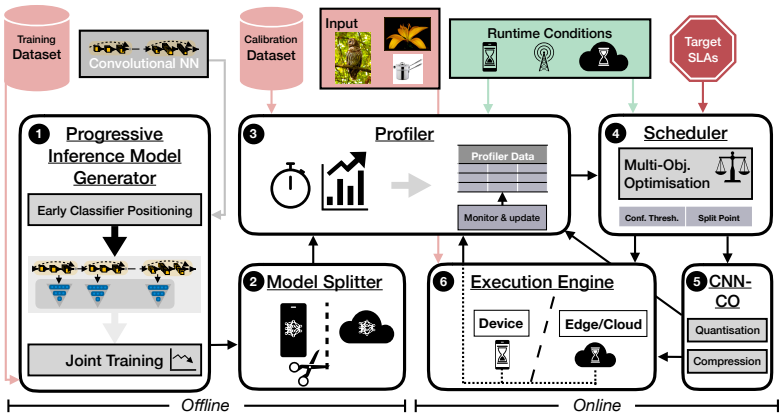

Figure 2: Overview of SPINN's architecture.

optimal partitioning scheme in the case of high and low server load. However, both systems only optimise for single-criterion objectives (latency or energy consumption), they lack support for app-specific SLAs, and suffer catastrophically when the remote server is unavailable. With a focus on data transfer, JALAD [47] incorporates lossy compression to minimise the offload transmission overhead. Nevertheless, to yield high performance, the proposed system sacrifices substantial accuracy (*i.e.* >5%). JointDNN [8] modelled CNN offloading as a graph split problem, but targets only offline scheduling and static environments instead of highly dynamic mobile settings. Contrary to these systems, SPINN introduces a novel scheduler that adapts the execution to the dynamic contextual conditions and jointly tunes the offloading point and early-exit policy to meet the application-level requirements. Moreover, by guaranteeing the presence of a local result, SPINN provides resilience to server disconnections.

Apart from offloading CNNs to a dedicated server, a number of works have focused on tangential problems. IONN [29] tackles a slightly different problem, where instead of preinstalling the CNN model to a remote machine, the client device can offload to any closeby server by transmitting both the incoming data and the model in a shared-nothing setup. Simultaneously, various works [50, 51, 85] have examined the case where the client device can offload to other devices in the local network. Last, [73] also employs cloud-device synergy and progressive inference, but with a very different focus, *i.e.* to perform joint classification from a multi-view, multi-camera standpoint. Its models, though, are statically allocated to devices and its fixed, statically-defined early-exit policy, renders it impractical for dynamic environments.

Offloading Multi-exit Models. Closer to our approach, Edgent [46] proposes a way of merging offloading with multi-exit models. Nonetheless, this work has several limitations. First, the inference workflow disregards data locality and always starts from the cloud. Consequently, inputs are always transmitted, paying an additional transfer cost. Second, early-exit networks are not utilised with progressive inference, *i.e.* inputs do not early-exit based on their complexity. Instead, Edgent tunes the model's complexity by selecting a *single* intermediary exit for all inputs. Therefore, the end system does not benefit from the variable complexity of inputs. Finally, the system has been evaluated solely on simple models (AlexNet) and datasets (CIFAR-10), less impacted by low-latency or unreliable network conditions. In contrast, SPINN exploits the fact that data already reside on the device to avoid wasteful input transfers, and employs a CNN-tailored technique to compress the offloaded data. Furthermore, not only our scheduler supports additional optimisation objectives, but it also takes advantage of the input's complexity to exit early, saving resource usage with minimal impact on accuracy.

3 PROPOSED SYSTEM

To remedy the limitations of existing systems, SPINN employs a progressive inference approach to alleviate the hard requirement for reliable device-server communication. The proposed system introduces a scheme of distributing progressive early-exit models across device and server, in which one exit is always present on-device, guaranteeing the availability of a result at all times. Moreover, as early exits along the CNN provide varying levels of accuracy, SPINN casts the acceptable prediction confidence as a tunable parameter to adapt its accuracy-speed trade-off. Alongside, we propose a novel run-time scheduler that jointly tunes the split point and early-exit policy of the progressive model, yielding a deployment tailored to the application performance requirements under dynamic conditions. Next, we present SPINN's high-level flow followed by a description of its components.

3.1 Overview

SPINN comprises offline components, run once before deployment, and online components, which operate at run time. Figure 2 shows a high-level view of SPINN. Before deployment, SPINN obtains a CNN model and derives a progressive inference network. This is accomplished by introducing early exits along its architecture and jointly training them using the supplied training set (Section 3.2 ❶). Next, the model splitter component (Section 3.3 ❷) identifies all candidate points in the model where computation can be split between device and cloud. Subsequently, the offline profiler (Section 3.4 ❸) calculates the exit-rate behaviour of the generated progressive model as well as the accuracy of each classifier. Moreover, it measures its performance on the client and server, serving as initial inference latency estimates.

At run time, the scheduler (Section 3.5 ❹) obtains these initial timings along with the target SLAs and run-time conditions and decides on the optimal split and early-exit policy. Given a split

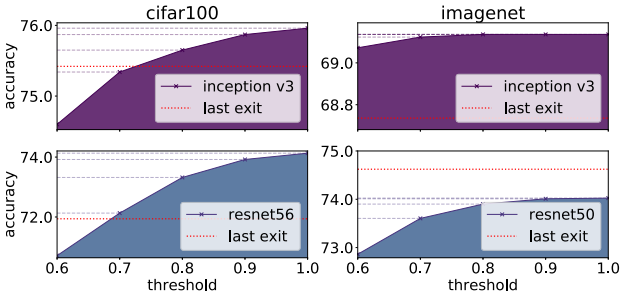

Figure 3: Accuracy of progressive networks across different confidence threshold values.

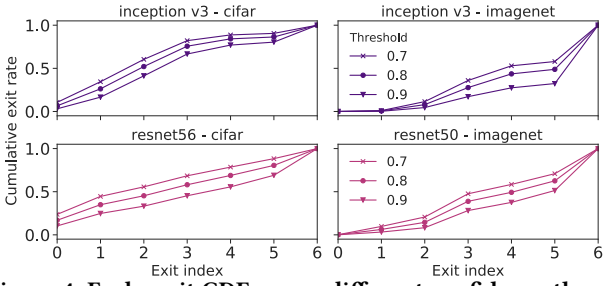

Figure 4: Early-exit CDF across different confidence threshold values.

point, the communication optimiser (Section 3.6 ❺) exploits the CNN's sparsity and resilience to reduced bitwidth to compress the data transfer and increase the bandwidth utilisation. The execution engine (Section 3.7 ❻) then orchestrates the distributed inference execution, handling all communication between partitions. Simultaneously, the online profiler monitors the execution across inferences, as well as the contextual factors (*e.g.* network, device/server load) and updates the initial latency estimates. This way, the system can adapt to the rapidly-changing environment, reconfigure its execution and maintain the same QoE.

3.2 Progressive Inference Model Generator

Given a CNN model, SPINN derives a progressive inference network (Figure 2 ❶). This process comprises a number of key design decisions: 1) the *number*, *position* and *architecture* of intermediate classifiers (early exits), 2) the *training scheme* and 3) the *early-exit policy*.

Early Exits. We place the intermediate classifiers along the depth of the architecture with equal distance in terms of FLOP count. With this platform-agnostic positioning strategy, we are able to obtain a progressive inference model that supports a wide range of latency budgets while being portable across devices. With respect to their number, we introduce six early exits in order to guarantee their convergence when trained jointly [35, 48], placed at 15%, 30%, … 90% of the network's total FLOPs. Last, we treat the architecture of the early exits as an invariant, adopting the design of [23], so that all exits have the same expressivity [59].

Training Scheme. We jointly train all classifiers from scratch and employ the cost function introduced in [35] as follows: $\mathcal{L} = \sum_{i=0}^{N-1} \tau_i * \mathcal{L}_i$ with τ_i starting uniformly at 0.01 and linearly increasing it to a maximum value of C_i, which is the relative position of the classifier in the network ($C_0 = 0.15, C_1 = 0.3, \ldots, C_{\text{final}} = 1$). The rationale behind this is to address the problem of "overthinking" [35], where some samples can be correctly classified by early exits while being misclassified deeper on in the network. This scheme requires the fixed placement of early exits prior to the training stage. Despite the inflexibility of this approach to search over different early-exit placements, it yields higher accuracy compared to the two-staged approach of training the main network and the early classifiers in isolation. In terms of training time, the end-to-end early-exit networks can take from 1.2× to 2.5× the time of the original network training, depending on the architecture and number of exits. In fact, the higher training overhead happens when the ratio of $\frac{FLOPS_{early\_classifier}}{FLOPS_{original\_network}}$ is higher. However, given that this cost is paid once offline, it is quickly amortised by the runtime latency benefits of early exiting on confident samples.

Early-exit Policy. For the early-exit strategy, we estimate a classifier's *confidence* for a given input using the top-1 output value of its softmax layer (Eq. (1)) [13]. An input takes the i-th exit if the prediction confidence is higher than a tunable threshold, thr_{conf} (Eq. (2)). The exact value of thr_{conf} provides a trade-off between the latency and accuracy of the progressive model and determines the *early-exit policy*. At run time, SPINN's scheduler periodically tunes thr_{conf} to customise the execution to the application's needs. If none of the classifiers reaches the confidence threshold, the most confident among them is used as the output prediction (Eq. (3)).

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad (\textit{Softmax of i-th exit}) \quad (1)$$

$$\arg_i \{\max_i \{\text{softmax}_i\} > thr_{\text{conf}}\} \quad (\textit{Check i-th exit's top-1}) \quad (2)$$

$$\underset{j \in \text{classifiers}}{\text{argmax}} \{\max_i \{\text{softmax}_i^j\}\} \quad (\textit{Return most confident}) \quad (3)$$

where z_i is the output of the final fully-connected layer for the i-th label, K the total number of labels, $j \in [0, 6]$ the classifier index and thr_{conf} the tunable confidence threshold.

Impact of Confidence Threshold. Figure 3 and 4 illustrate the impact of different *early-exit policies* on the accuracy and early-exit rate of progressive models, by varying the confidence threshold (thr_{conf}). Additionally, Figure 3 reports on the accuracy without progressive inference (*i.e.* last exit only, represented by the red dotted line). Note that exiting only at the last exit can lead to lower accuracy than the progressive models for some architectures, a phenomenon that can be attributed to the problem of "overthinking"[3].

Based on the figures, we draw two major conclusions that guide the design of SPINN's scheduler. First, across all networks, we observe a monotonous trend with higher thresholds leading to higher accuracies (Figure 3) while lower ones lead to more samples exiting earlier (Figure 4). This exposes the confidence threshold as a tunable parameter to control accuracy and overall processing time. Second, different networks behave differently, producing *confident* predictions at different exits along the architecture. For example, Inception-v3 on CIFAR-100 can obtain a confident prediction earlier on, whereas ResNet-50 on ImageNet cannot classify robustly from early features only. In this respect, we conclude that optimising the confidence threshold for each CNN explicitly is key for tailoring the deployment to the target requirements.

[3] "Overthinking" [35] dictates that certain samples that would normally get misclassified by reaching the final classifier of the network if they exit early, they get classified correctly. This leads to small accuracy benefits of progressive inference networks that neither the original model would have (due to early-exiting) nor a single-exit smaller variant (due to late-exiting).

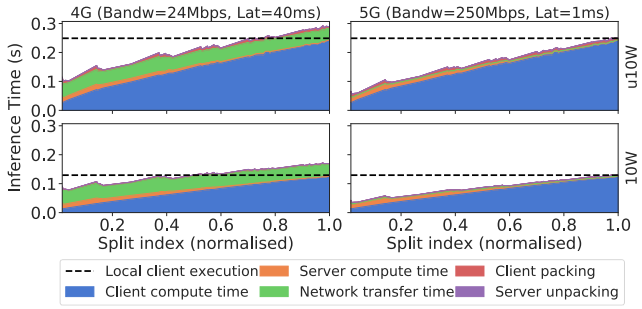

Figure 5: ResNet-56 inference times for different network conditions (4G and 5G) and client device compute capabilities (Jetson 10W and u10W), when offloading to the cloud without early exits.

3.3 Model Splitter

After deriving a progressive model from the original CNN, SPINN aims to split its execution across a client and a server in order to dynamically utilise remote resources as required by the application SLAs. The model splitter (Figure 2 ❷) is responsible for 1) defining the potential split points and 2) identifying them automatically in the given CNN.

Split Point Decision Space. CNNs typically consist of a sequence of layers, organised in a feed-forward topology. SPINN adopts a partition scheme which allows splitting the model along its depth at layer granularity. For a CNN with N_L layers, there are N_L-1 candidate points, leading to 2^{N_L-1} possible partitions. To reduce the search space and minimise the number of transmissions across the network, we make two key observations. First, since CNN final outputs are rather small, once execution is offloaded to the powerful remote server, there is no gain in having two or more split points as this would incur in extra communication costs. Second, different layer splits have varying transmission costs and compression potential. For example, activation layers such as ReLU [54] cap negative values to zero, which means that their output becomes more compressible [56, 60, 82] and they can be more efficiently transferred by SPINN's *communication optimiser* (Section 3.6). Therefore, while SPINN's design supports an arbitrary number of split points and layers, in this work, we allow one split point per CNN and reduce the candidate split points to ReLU layers.

Automatic Split Point Identification. To automatically detect all candidate split points in the given CNN, the model splitter employs a dynamic analysis approach. This is performed by first constructing the execution graph of the model in the target framework (*e.g. PyTorch*), identifying all split points and the associated dependencies, and then applying SPINN's partitioning scheme to yield the final pruned split point space. The resulting set of points defines the allowed partitions that can be selected by the scheduler (Section 3.5).

Impact of Split Point. To investigate how the split point selection affects the overall latency, we run multiple CNN splits between a Nvidia Jetson Xavier client and a server (experimental setup detailed in Section 4). Figure 5 shows the breakdown of ResNet-56's inference times with CIFAR-100 over distinct network conditions and client compute capabilities - u10W and 10W. Packing refers to the runtime of the communication optimiser module, detailed in Section 3.6.

Based on the figure, we make two observations. First, different split points yield varying trade-offs in client, server and transfer time. For example, earlier splits execute more on the server, while later ones execute more on-device but often with smaller transfer requirements[4]. This indicates that SPINN's scheduler can selectively choose one to minimise any given runtime (*e.g.* device, server or transfer), as required by the user. Second, dynamic conditions such as the connectivity, and the device compute capabilities play an important role in shaping the split point latency characteristics illustrated in Figure 5. For example, a lower-end client (u10W) or a loaded server would require longer to execute its allocated split, while low bandwidth can increase the transfer time. This indicates that it is hard to statically identify the best split point and highlights the need for an informed partitioning that adapts to the environmental conditions in order to meet the application-level performance requirements.

3.4 Profiler

Given the varying trade-offs of different split points and confidence thresholds, SPINN considers the client and server load, the networking conditions and the expected accuracy in order to select the most suitable configuration. To estimate this set of parameters, the *profiler* (Figure 2 ❸) operates in two stages: i) offline and ii) run-time.

Offline stage: In the offline phase, the profiler performs two kinds of measurements, *device-independent* and *device-specific*. The former include CNN-specific metrics, such as 1) the size of data to be transmitted for each candidate split and 2) the average accuracy of the progressive CNN for different confidence thresholds. These are measured only once prior to deployment. Next, the profiler needs to obtain latencies estimates that are specific to each device. To this end, the profiler measures the average execution time per layer by passing the CNN through a calibration set – sampled from the validation set of the target task. The results serve as the initial latency and throughput estimates.

Run-time stage: At run time, the profiler refines its offline estimates by regularly monitoring the device and server load, as well as the connectivity conditions. To keep the profiler lightweight, instead of adopting a more accurate but prohibitively expensive estimator, we employ a 2-staged linear model to estimate the inference latency on-device.

In the first step, the profiler measures the actual on-device execution time up to the split point s, denoted by $T^{\text{real}}_{\langle s \rangle}$ during each inference. Next, it calculates a latency scaling factor SF as the ratio between the actual time and the offline latency estimate up to the split s, *i.e.* $SF = \frac{T^{\text{real}\langle s \rangle}}{T^{\text{offline}\langle s \rangle}}$. As a second step, the profiler treats the scaling factor as an indicator of the load of the client device, and uses it to estimate the latency of all other candidate splits. Thus, the latency of a different split s' is estimated as $SF \cdot T^{\text{offline}}_{\langle s' \rangle}$.

[4]Note that independently of the amount of transmitted data, there is always a network latency overhead that must be amortised, which in the case of 4G, is quite significant.

Algorithm 1: Flow of dynamic scheduler upon invocation

Input: Space of candidate designs Σ
Prioritised hard constraints $\langle C_1, C_2, ..., C_n \rangle$
Prioritised soft targets $\langle O_1, O_2, ..., O_{|\mathcal{M}|} \rangle$
Current network conditions $net = \langle L, B \rangle$
Current device and server loads $l^{\{\text{dev,server}\}}$
Profiler data prf
Output: Highest performing design $\sigma^* = \langle s^*, thr^*_{\text{conf}} \rangle$

1 $prf \leftarrow$ UpdateTimings(prf, net, l^{dev}, l^{server})
2 $\Sigma^{\text{feasible}} \leftarrow \Sigma$
3 /* - - - *Obtain feasible space based on hard constraints* - - -*/
4 **foreach** $C_i \in \langle C_1, C_2, ..., C_n \rangle$ **do**
5 $\quad \Sigma^{\text{feasible}} \leftarrow$ RemoveInfeasiblePoints(prf, C_i, Σ^{feasible})
6 $\quad$ ↳ **VecCompare(**prf, $\Sigma^{\text{feasible}}(:,M_i)$, op_i, thr_i**)** $\forall i \in [1, n]$
7 **end**
8 /* - - - *Optimise user-defined metrics - Eq. (4)* - - -*/
9 $\sigma^* \leftarrow$ OptimiseUserMetrics(prf, $\langle O_1, O_2, ..., O_{|\mathcal{M}|} \rangle$, Σ^{feasible})
10 ↳ **VecMax/Min(**prf, $\Sigma^{\text{feasible}}(:,M_i)$, op_i**)** $\forall i \in [1, |\mathcal{M}|]$

Similarly, to assess the server load, the remote endpoint's compute latency is optionally communicated back to the device, piggybacked with the CNN response when offloading. If the server does not communicate back latencies for preserving the privacy of the provider, these can be coarsely estimated as $T^{\text{server}}_{\langle s \rangle} = T^{\text{response}}_{\langle s,e \rangle} - \left(L + \frac{D_{\text{response}}}{B}\right)$, where $T^{\text{response}}_{\langle s \rangle}$ is the total time for the server to respond with the result for split point s and exit e, D_{response} is the size of transferred data and B, L are the instantaneous network bandwidth and latency respectively. We periodically offload to the server without stopping the local execution to reassess when the server transitions from "overloaded" to "accepting requests".

To estimate the instantaneous *network transfer* latency, the profiler employs a run-time monitoring mechanism of the bandwidth B and latency L experienced by the device [15]. The overall transfer time is $L + \frac{D_{\langle s \rangle}}{B}$, where $D_{\langle s \rangle}$ is the amount of data to be transferred given split s. As the network conditions change, the monitoring module refines its estimates by means of two moving averages: a real-time estimation (L^{rt}, B^{rt}) and a historical moving average (L^{hist}, B^{hist}). The former is updated and used only when transfers have occurred within the last minutes. If no such information exists, the historical estimates for the same network type are used.

3.5 Dynamic Scheduler

Given the output of the profiler, the dynamic scheduler (Figure 2 ❹) is responsible for distributing the computation between device and cloud, and deciding the early-exit policy of the progressive inference network. Its goal is to yield the highest performing configuration that satisfies the app requirements. To enable the support of realistic multi-criteria SLAs, the scheduler incorporates a combination of *hard constraints* (*e.g.* a strict inference latency deadline of 100 ms) and *soft targets* (*e.g.* minimise cost on the device side). Internally, we capture these under a multi-objective optimisation (MOO) formulation. The current set of metrics is defined as

$$\mathcal{M} = \{latency, throughput, server\ cost, device\ cost, accuracy\}$$

In SPINN, we interpret cloud and device cost as the execution time on the respective side. The defined metrics set $\mathcal{M}$, together with the associated constraints, can cover a wide range of use-cases, based on the relative importance between the metrics for the target task. Formally, we define a hard constraint as $C = \langle M, op, thr \rangle$ where $M \in \mathcal{M}$ is a metric, op is an operator, *e.g.* $\leq$, and thr is a threshold value.

With respect to soft optimisation targets, we define them formally as $O = \langle M, min/max/value \rangle$ where a given metric $M \in \mathcal{M}$ is either maximised, minimised or as close as possible to a desirable value. To enable the user to specify the importance of each metric, we employ a multi-objective lexicographic formulation [52], shown in Eq. (4).

$$\min_{\sigma} M_i(\sigma), \text{ s.t. } M_j(\sigma) \leq M_j(\sigma^*_j) \qquad (4)$$
$$j = 1, 2, ..., i-1\ ,\ i > 1\ ,\ i = 1, 2, ..., |\mathcal{M}|$$

where σ represents a design point, $M_i \in \langle M_1, M_2, ..., M_{|\mathcal{M}|} \rangle$ is the i-th metric in the ordered tuple of soft targets, i is a metric's position in the importance sequence and $M_j(\sigma^*_j)$ represents the optimum of the j-th metric, found in the j-th iteration. Under this formulation, the user ranks the metrics in order of importance as required by the target use-case.

Algorithm 1 presents the scheduler's processing flow. As a first step, the scheduler uses the estimated network latency and bandwidth, and device and server loads to update the profiler parameters (line 1), including the attainable latency and throughput, and device and server cost for each candidate configuration. Next, all infeasible solutions are discarded based on the supplied hard constraints (lines 4-7); given an ordered tuple of prioritised constraints $\langle C_1, C_2, ..., C_n \rangle$, the scheduler iteratively eliminates all configurations σ=$\langle s, thr_{\text{conf}} \rangle$ that violate them in the given order, where s and thr_{conf} represent the associated split point and confidence threshold respectively. In case there is no configuration to satisfy all the constraints up to i+1, the scheduler adopts a best-effort strategy by keeping the solutions that comply with up to the i-th constraint and treating the remaining *n-i* constraints as soft targets. Finally, the scheduler performs a lexicographic optimisation of the user-prioritised soft targets (lines 9-10). To determine the highest performing configuration σ^*, the scheduler solves a sequence of $|\mathcal{M}|$ single-objective optimisation problems, *i.e.* one for each $M \in \langle M_1, M_2, ..., M_{|\mathcal{M}|} \rangle$ (Eq. (4)).

Deployment. Upon deployment, the scheduler is run on the client side, since most relevant information resides on-device. In a multi-client setting, this setup is further reinforced by the fact that each client device decides independently on its offloading parameters. However, to be deployable without throttling the resources of the target mobile platform, the scheduler has to yield low resource utilisation at run time. To this end, we vectorise the comparison, maximisation and minimisation operations (lines 5-6 and 9-10 in Algorithm 1) to utilise the SIMD instructions of the target mobile CPU (*e.g.* the NEON instructions on ARM-based cores) and minimise the scheduler's runtime.

At run time, although the overhead of the scheduler is relatively low, SPINN only re-evaluates the candidate configurations when the outputs of the profiler change by more than a predefined threshold. For highly transient workloads, we can switch from a moving average to an exponential back-off threshold model for mitigating too many scheduler calls. The scheduler overhead and the tuning of the invocation frequency is discussed in Section 4.3.2.

The server – or HA proxy[5] [70] in multi-server architectures – can admit and schedule requests on the remote side to balance the workload and minimise inference latency, maximise throughput or minimise the overall cost by dynamically scaling down unused

[5]High-Availability proxy for load balancing & fault tolerance in data centres.

resources. We consider these optimisations cloud-specific and out of the scope of this paper. As a result, in our experiments we account for a single server always spinning and having the model resident to its memory. Nevertheless, in a typical deployment, we would envision a caching proxy serving the models with RDMA to the CPU or GPU of the end server, in a virtualised or serverless environment so as to tackle the cold-start problem [57, 77]. Furthermore, to avoid oscillations (flapping) of computation between the deployed devices and the available servers, techniques used for data-center traffic flapping are employed [5].

3.6 CNN Communication Optimiser

CNN layers often produce large volumes of intermediate data which come with a high penalty in terms of network transfer. A key enabler in alleviating the communication bottleneck in SPINN is the communication optimiser module (CNN-CO) (Figure 2 ❺). CNN-CO comprises two stages. In the first stage, we exploit the resilience of CNNs to low-precision representation [14, 16, 28, 38] and lower the data precision from 32-bit floating-point down to 8-bit fixed-point through linear quantisation [28, 53]. By reducing the bitwidth of *only* the data to be transferred, our scheme allows the transfer size to be substantially lower without significant impact on the accuracy of the subsequent classifiers (*i.e.* <0.65 percentage point drop across all exits of the examined CNNs). Our scheme differs from both i) *weights-only reduction* [17, 67, 86], which minimises the model size rather than activations' size and ii) *all-layers quantisation* [14, 16, 24, 28] which requires complex techniques, such as quantisation-aware training [24, 28] or a re-training step [14, 16], to recover the accuracy drop due to the precision reduction across all layers.

The second stage exploits the observation that activation data are amenable to compression. A significant fraction of activations are zero-valued, meaning that they are sparse and highly compressible. As noted by prior works [56, 60, 82], this sparsity of activations is due to the extensive use of the ReLU layer that follows the majority of layers in modern CNNs. In CNN-CO, sparsity is further magnified due to the reduced precision. In this respect, CNN-CO leverages the sparsity of the 8-bit data by means of an LZ4 compressor with bit shuffling.

At run time, SPINN predicts whether the compression cost will outweigh its benefits by comparing the estimated CNN-CO runtime to the transfer time savings. If CNN-CO's overhead is amortised, SPINN queues offloading requests' data to the CNN-CO, with dedicated threads for each of the two stages, before transmission. Upon reception at the remote end, the data are decompressed and cast back to the original precision to continue inference. The overhead is shown as *packing* in Figure 5 for non-progressive models.

3.7 Distributed Execution Engine

In popular Deep Learning frameworks, such as *TensorFlow* [1] and *PyTorch* [58], layers are represented by *modules* and data in the form of multi-dimensional matrices, called *tensors*. To split and offload computation, SPINN modifies CNN layer's operations behind the scenes. To achieve this, it intercepts module and tensor operations by replacing their functions with a custom wrapper using Python's function decorators.

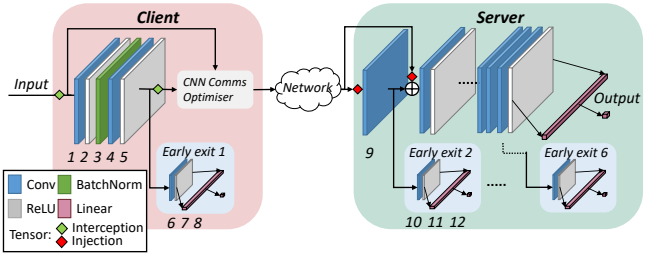

Figure 6: Offloading a progressive ResNet block.

Figure 6 focuses on an instance of an example ResNet block. SPINN attributes IDs to each layer in a trace-based manner, by executing the network and using the layer's execution order as a sequence identifier.[6] SPINN uses these IDs to build an execution graph in the form of a directed acyclic graph (DAG), with nodes representing tensor operations and edges the tensor flows among them [1, 62, 75]. This is then used to detect the dependencies across partitions. To achieve this, Python's dynamic instance attribute registration is used to `taint` tensors and monitor their flow through the network. With Figure 6 as a reference, SPINN's custom wrapper (Figure 2 ❻) performs the following operations:

Normal execution: When a layer is to be run locally, the wrapper calls the original function it replaced. In Figure 6, layers 1 to 8 execute normally on-device, while layers from 9 until the end execute normally on the server side.

Offload execution: When a layer is selected as a partition point (layer 9), instead of executing the original computation on the client, the wrapper queues an offloading request to be transmitted. This request contains the inputs (layer 5's output) and subsequent layer dependencies (input of skip connection). Furthermore, the inference and the data transfer are decoupled in two separate threads, to allow for pipelining of data transfer and next frame processing.

Resume execution: Upon receiving an offloading request, the inputs and dependencies are injected in the respective layers (layer 9 and *add* operation) and normal execution is resumed on the remote side. When the server concludes execution, the results are sent back in a parallel thread.

Early exit: When an intermediate classifier (*i.e.* early exit) is executed, the wrapper evaluates its prediction confidence (Eq. (1)). If it is above the provided thr_{conf}, execution terminates early, returning the current prediction (Eq. (2)).

Since the premise of our system is to always have at least one usable result on the client side, we continue the computation on-device even past the split layer, until the next early exit is encountered. Furthermore, to avoid wasteful data transmission and redundant computation, if a client-side early exit covers the latency SLA and satisfies the selected thr_{conf}, the client sends a termination signal to the remote worker to cancel the rest of the inference. If remote execution has not started yet, SPINN does not offload at all.

4 EVALUATION

This section presents the effectiveness of SPINN in significantly improving the performance of mobile CNN inference by examining its core components and comparing with the currently standard device-

[6] Despite the existence of branches, CNN execution tends to be parallelised across data rather than layers. Hence, the numbering is deterministic.

Platform	CPU	Clock Freq.	Memory	GPU
Server	2× Intel Xeon Gold 6130	2.10 GHz	128 GB	GTX1080Ti
Jetson AGX	Carmel ARMv8.2	2.26 GHz	16 GB	512-core Volta

Table 1: Specifications of evaluated platforms.

and cloud-only implementations and state-of-the-art collaborative inference systems.

4.1 Experimental Setup

For our experiments, we used a powerful computer as the server and an Nvidia Jetson Xavier AGX as the client (Table 1). Specifically for Jetson, we tested against three different power profiles to emulate end-devices with different compute capabilities:[7] 1) *30W* (full power), 2) *10W* (low power), 3) *underclocked 10W* (u10W). Furthermore, to study the effect of limited computation capacity (*e.g.* high-load server), we emulated the load by linearly scaling up the CNN computation times on the server side. We simulated the network conditions of offloading by using the average bandwidth and latency across national carriers [26, 27], for 3G, 4G and 5G mobile networks. For local-area connections (Gigabit Ethernet 802.3, WiFi-5 802.11ac), we used the nominal speeds of the protocol. We have developed SPINN on top of *PyTorch* (1.1.0) and experimented with four models, altered from *torchvision* (0.3.0) to include early exits or to reflect the CIFAR-specific architectural changes. We evaluated SPINN using: ResNet-50 and -56 [20], VGG16 [63], MobileNetV2 [61] and Inception-v3 [69]. Unless stated otherwise, each benchmark was run 50 times to obtain the average latency.

Datasets and Training. We evaluated SPINN on two datasets, namely CIFAR-100 [41] and ImageNet (ILSVRC2012) [10]. The former contains 50k training and 10k test images of resolution 32×32, each corresponding to one of 100 labels. The latter is significantly larger, with 1.2m training and 50k test images of 300×300 and 1000 labels. We used the preprocessing steps described in each model's implementation, such as *scaling* and *cropping* the input image, stochastic image *flip* ($p = 0.5$) and colour channel *normalisation*. After converting these models to progressive early-exit networks, we trained them jointly from scratch end-to-end, with the "overthink" cost function (Section 3.2). We used the authors' training hyperparameters, except for MobileNetV2, where we utilised SGD with learning rate of 0.05 and cosine learning rate scheduling, due to convergence issues. We trained the networks for 300 epochs on CIFAR-100 and 90 epochs on ImageNet.

4.2 Performance Comparison

This subsection presents a performance comparison of SPINN with: 1) the two state-of-the-art CNN offloading systems Neurosurgeon [34] and Edgent [46] (Section 2); 2) the status-quo cloud- and device-only baselines; and 3) a non-progressive ablated variant of SPINN.

4.2.1 Throughput Maximisation. Here, we assess SPINN's inference throughput across varying network conditions. For these experiments, the SPINN scheduler's objectives were set to maximise throughput with up to 1 percentage point (pp) tolerance in accuracy drop with respect to the CNN's last exit.

[7] We are adjusting the TDP and clock frequency of the CPU and GPU cores, effectively emulating different tiers of devices, ranging from high-end embedded devices to mid-tier smartphones.

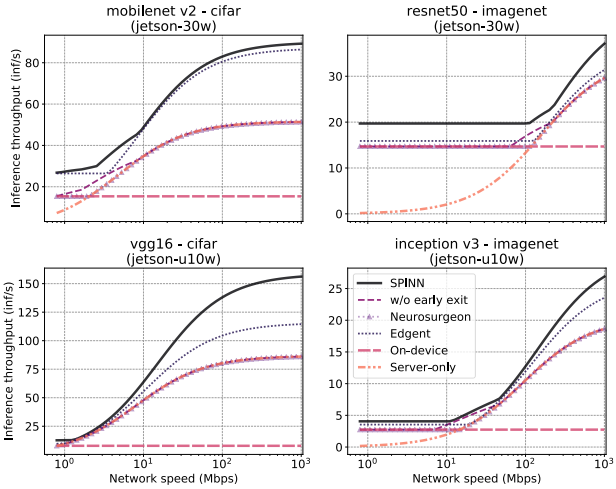

Figure 7: Achieved throughput for various ⟨model, device, dataset⟩ setups vs. network speed.

Figure 7 shows the achieved inference throughput for varying network speeds. On-device execution yields the same throughput independently of the network variation, but is constrained by the processing power of the client device. Server-only execution follows the trajectory of the available bandwidth. Edgent always executes a part of the CNN (up to the exit that satisfies the 1pp accuracy tolerance) irrespective of the network conditions. As a result, it follows a similar trajectory to server-only but achieves higher throughput due to executing only part of the model. Neurosurgeon demonstrates a more polarised behaviour; under constrained connectivity it executes the whole model on-device, whereas as bandwidth increases it switches to offloading all computation as it results in higher throughput. The ablated variant of SPINN (*i.e.* without early exits) largely follows the behaviour of Neurosurgeon at the two extremes of the bandwidth while in the middle range, it is able to achieve higher throughput by offloading earlier due to CNN-CO compressing the transferred data.

The end-to-end performance achieved by SPINN delivers the highest throughput across all setups, achieving a speedup of up to 83% and 52% over Neurosurgeon and Edgent, respectively. This can be attributed to our bandwidth- and data-locality-aware scheduler choices on the early-exit policy and partition point. In low bandwidths, SPINN selects device-only execution, outperforming all other on-device designs due to its early-exiting mechanism, tuned by the scheduler module. In the mid-range, the CNN-CO module enables SPINN to better utilise the available bandwidth and start offloading earlier on, outperforming both Edgent and Neurosurgeon. In high-bandwidth settings, our system surpasses the performance of all other designs by exploiting its optimised early-exiting scheme.

Specifically, compared to Edgent, SPINN takes advantage of the input's classification difficulty to exit early, whereas the latter only selects an intermediate exit to *uniformly* classify all incoming samples. Moreover, in contrast with Edgent's strategy to always transmit the input to the remote endpoint, we exploit the fact that data already reside on the device and avoid the wasteful data transfers.

4.2.2 Server-Load Variation. To investigate the performance of SPINN under various server-side loads, we measured the inference

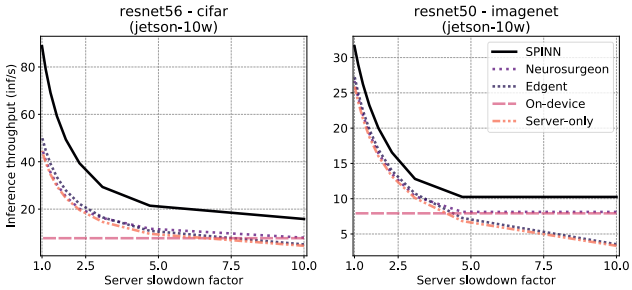

Figure 8: Effect of server slowdown on ResNet.

throughput of SPINN against baselines when varying the load of the remote end, with 1pp of accuracy drop tolerance. This is accomplished by linearly scaling the latency of the server execution by a slowdown factor (*i.e.* a factor of 2 means the server is 2× slower). Figure 8 presents the throughput achieved by each approach under various server-side loads, with the Jetson configured at the 10W profile and the network speed in the WiFi-5 range (500 Mbps).

With low server load (left of the x-axis), the examined systems demonstrate a similar trend to the high-bandwidth performance of Figure 7. As the server becomes more loaded (*i.e.* towards the right-hand side), performance deteriorates, except for the case of device-only execution which is invariant to the server load. On the one hand, although its attainable throughput reduces, Neurosurgeon adapts its policy based on the server utilisation and gradually executes a greater fraction of the CNN on the client side. On the other hand, Edgent's throughput deteriorates more rapidly and even reaches below the device-only execution under high server load, since its run-time mechanism does not consider the varying server load. Instead, by adaptively optimising both the split point and the early-exit policy, SPINN's scheduler manages to adapt the overall execution based on the server-side load, leading to throughput gains between 1.18-1.99× (1.57× geo. mean) and 1.15-3.09× (1.61× geo. mean) over Neurosurgeon and Edgent respectively.

4.2.3 Case Study: Latency-driven SLAs at minimal server cost. To assess SPINN's performance under deadlines, we target the scenario where a service provider aims to deploy a CNN-based application that meets strict latency SLAs at maximum accuracy and minimal server-side cost. In this setting, we compare against Neurosurgeon[8] and Edgent, targeting MobileNetV2 and ResNet-56 over 4G. Figure 9 shows the server computation time and accuracy achieved by each system for two device profiles with different compute capabilities - u10W and 10W. Latency SLAs are represented as a percentage of the device-only runtime of the original CNN (*i.e.* 20% SLA means that the target is 5× less latency than on-device execution, requiring early exiting and/or server support).

For the low-end device (u10W) and strict latency deadlines, SPINN offloads as much as possible to the cloud as it allows reaching faster a later exit in the network, hence increasing the accuracy. As the SLA loosens (reaching more than 40% of the on-device latency), SPINN starts to gradually execute more and more locally. In contrast, Edgent and Neurosurgeon achieve similar accuracy but with up to 4.9× and 6.8× higher server load. On average across all targets, SPINN reduces Edgent and Neurosurgeon server times by

[8]It should be noted that Neurosurgeon maintains the accuracy of the original CNN.

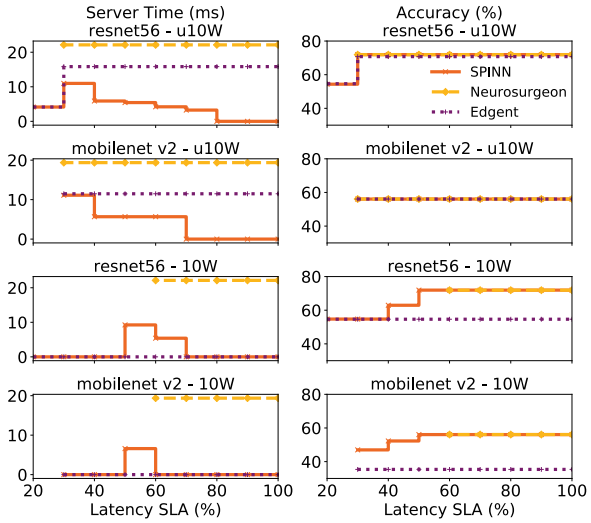

Figure 9: Server time (left) and accuracy (right) of SPINN vs. Neurosurgeon and Edgent for different latency SLAs and client compute power (u10W and 10W). The SLA is expressed as a percentage of the on-device latency.

68.64% and 82.5% (60.3% and 83.6% geo. mean), respectively, due to its flexible multi-objective scheduler. Instead, Neurosurgeon can only optimise for overall latency and cannot trade off accuracy to meet the deadline (*e.g.* for 20% SLA on ResNet-56) while Edgent cannot account for server-time minimisation and accuracy drop constraints.

The situation is different for the more powerful device (10W). With the device being faster, the SLA targets become much stricter. Therefore, we observe that SPINN and Edgent can still meet a latency constraint as low as 20% and 30% of the local execution time for ResNet-56 and MobileNetV2 respectively. In contrast, without progressive inference, it is impossible for Neurosurgeon to achieve inference latency below 60% of on-device execution across both CNNs. In this context, SPINN is able to trade off accuracy in order to meet stricter SLAs, but also improve its attainable accuracy as the latency constraints are relaxed.

For looser latency deadlines (target larger than 50% of the on-device latency), SPINN achieves accuracy gains of 17.3% and 20.7% over Edgent for ResNet-56 and MobileNetV2, respectively. The reason behind this is twofold. First, when offloading, Edgent starts the computation on the server side, increasing the communication latency overhead. Instead, SPINN's client-to-server offloading strategy and compression significantly reduces the communication latency overhead. Second, due to Edgent's unnormalised cost function (*i.e.* $\max\left(\frac{1}{lat} + acc\right)$), the throughput's reward dominates the accuracy gain, leading to always selecting the first early-exit sub-network and executing it locally. In contrast, SPINN's scheduler's multi-criteria design is able to capture accuracy, server time and latency constraints to yield an optimised deployment. Hence, similarly to the slower device, SPINN successfully exploits the server resources to boost accuracy under latency constraints, while it can reach up to pure on-device execution for loose deadlines.

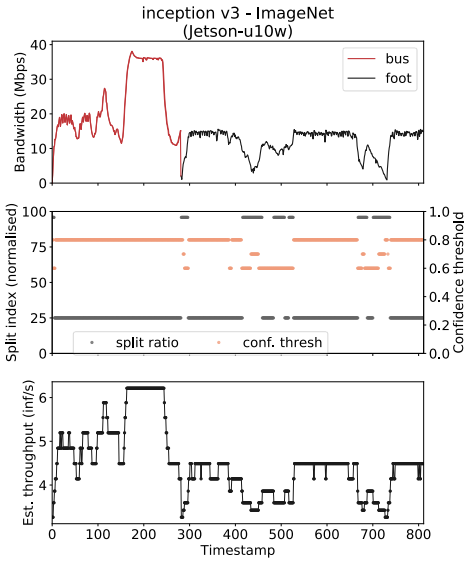

Figure 10: SPINN scheduler's behaviour on real network provider trace.

4.3 Runtime Overhead and Efficiency

4.3.1 Deployment Overhead. By evaluating across our examined CNNs and datasets on the CPU of Jetson, the scheduler executes in max 14 ms (11 ms geo. mean). This time includes the cost of reading the profiler parameters, updating the monitored metrics, and searching for and returning the selected configuration. Moreover, SPINN's memory consumption is in the order of a few KB (*i.e.* <1% of Jetson's RAM). These costs are amortised over multiple inferences, as the scheduler is invoked only on significant context changes. We discuss the selection of such parameters in the following section.

4.3.2 Network Variation. To assess the responsiveness of SPINN in adapting to dynamic network conditions, we targeted a real bandwidth trace from a Belgian ISP. The trace contains time series of network bandwidth variability during different user activities. In this setup, SPINN executes the ImageNet-trained Inception-v3 with Jetson-u10W as the client under the varying bandwidth emulated by the Belgium 4G/LTE logs. The scheduler is configured to maximise both throughput and accuracy. Figure 10 (top) shows an example bandwidth trace from a moving bus followed by walking. Figure 10 (bottom) shows SPINN's achieved inference throughput under the changing network quality. The associated scheduler decisions are depicted in Figure 10 (middle).

At low bandwidth (<5 Mbps), SPINN falls back to device-only execution. In these cases, the scheduler adopts a less conservative early-exit policy by lowering the confidence threshold. In this manner, it allows more samples to exit earlier, compensating for the client's low processing power. Nonetheless, the impact on accuracy remains minimal (<1%) for the selected early-exit policies by the scheduler ($thr_{\text{conf}} \in [0.6, 1.0]$), as illustrated in Figure 3 for Inception-v3 on ImageNet. At the other end, high bandwidths result in selecting an earlier split point and thus achieving up to 7× more inf/sec over pure on-device execution. Finally, the similar trajectories of the top and bottom figure suggest that our scheduler can adapt the system to the running conditions, without having to be continuously invoked.

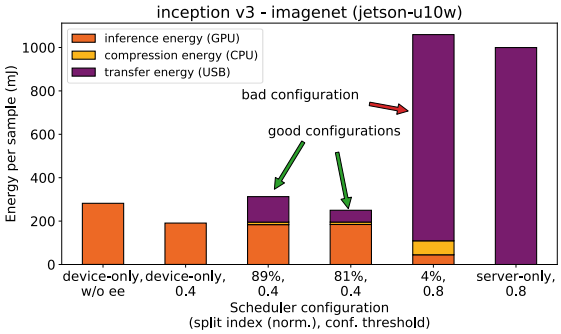

Figure 11: Energy consumption of SPINN vs. baselines.

Overall, we observe that small bandwidth changes do not cause significant alterations to the split and early-exit strategies. By employing an averaging historical window of three values and a difference threshold of 5%, the scheduler is invoked 1/3 of the total bandwidth changes across traces.

4.4 Energy Consumption

Figure 11 shows the breakdown of dominant energy consumption across the client device subsystems. We measured energy consumption over 1000 inferences from the validation set and offloading over UK's Three Broadband's 4G network with a Huawei E3372 USB adapter. We measured the power of Jetson (CPU, GPU) from its integrated probes and the transmission energy with the Monsoon AAA10F power monitor.

Traversing the horizontal axis left-to-right, we first see device-only execution without and with early-exits, where the local processing dominates the total energy consumption. The latter shows benefits due to samples exiting early from the network. Next, we showcase the consumption breakdown of three different $\langle split, thr_{\text{conf}} \rangle$ configurations. The first two configurations demonstrate comparable energy consumption with the device-only execution without early exits. On the contrary, a bad configuration requires an excessively large transfer size, leading to large compression and transfer energy overheads. Last, the energy consumption when fully offloading is dominated by the network transfer.

Across configurations, we witness a 5× difference in energy consumption across different inference setups. While device-only execution yields the lowest energy footprint per sample, it is also the slowest. Our scheduler is able to yield deployments that are significantly more energy efficient than full offloading (4.2×) and on par with on-device execution (0.76 – 1.12×), while delivering significantly faster end-to-end processing. Finally, with different configurations varying both in energy and performance, the decision space is amenable to energy-driven optimisation by adding energy as a scheduler optimisation metric.

4.5 Constrained Availability Robustness

Next we evaluate SPINN's robustness under constrained availability of the remote end such as network timeouts, disconnections and server failures. More specifically, we investigate 1) the achieved accuracy across various failure rates and 2) the latency improvement over conventional systems enhanced with an error-control policy. In these experiments, we fix the confidence threshold of three models (Inception-v3, ResNet-56 and ResNet-50) to a value of 0.8

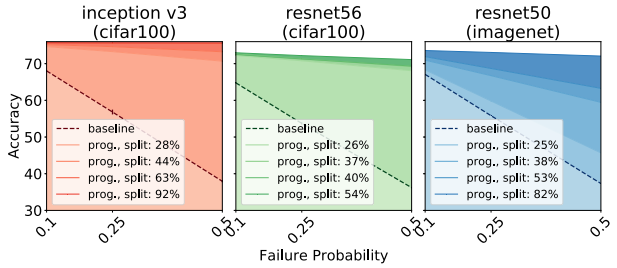

Figure 12: Comparison of the average accuracy under uncertain server availability. The shaded area indicates attained accuracies under a probability distribution.

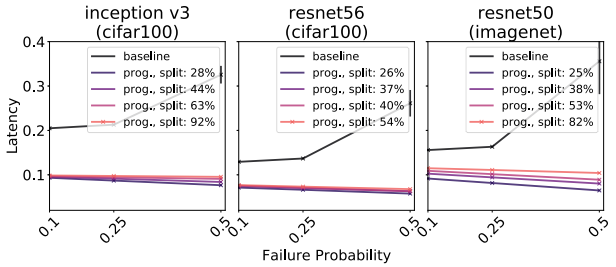

Figure 13: Comparison of average latency under uncertain server availability.

and emulate variable failure rates by sampling from a random distribution across the validation set.

Accuracy comparison: For this experiment (Figure 12), we compare SPINN at different network split points (solid colours) against a non-progressive baseline (dashed line). Under failure conditions, the baseline unavoidably misclassifies the result as there is no usable result locally on-device. However, SPINN makes it possible to acquire the most confident local result up to the split point, when the server is unavailable.

As shown in Figure 12, the baseline quickly drops in accuracy as the failure rate increases. This is not the case with SPINN, which manages to maintain a minimal accuracy drop. Specifically, we witness drops ranging in $[0, 5.75\%]$ for CIFAR-100 and $[0.46\%, 33\%]$ for ImageNet, when the equivalent drop of the baseline is $[11.56\%, 44.1\%]$ and $[9.25\%, 44.34\%]$, respectively. As expected, faster devices are able to execute locally a larger part of the model (exit later) while meeting their SLA exhibit the smallest impact under failure, as depicted in the progressive variants of the two models.

Latency comparison: In this scenario, we compare SPINN against a single-exit offloaded variant of the same networks. This time instead of simply failing the inference when the remote end is unavailable, we allow for *retransmission* with *exponential back-off*, a common behaviour of distributed systems to avoid channel contention. When a sample fails under the respective probability distribution, the result gets retransmitted. If the same sample fails again, the client waits double the time and retransmits, until it succeeds. Here, we assume Jetson-10W offloads to our server over 4G and varying failure probability ($P_{\text{fail}} \in \{0.1, 0.25, 0.5\}$). The initial retransmission latency is 20 ms. We ran each experiment three times and report the mean and standard deviation of the latencies.

As depicted in Figure 13, the non-progressive baseline follows a trajectory of increasing latency as the failure probability gets higher, due to the additional back-off latency each time a sample fails. While the impact on the average latency for both networks going from $P_{\text{fail}} = 0.1$ to $P_{\text{fail}} = 0.25$ is gradual, at 3.9%, 5.8% and 4.7% for Inception-v3, ResNet-56 and ResNet-50 respectively, the jump from $P_{\text{fail}} = 0.25$ to $P_{\text{fail}} = 0.5$ is much more dramatic, at 52.9%, 91% and 118%. The variance at $P_{\text{fail}} = 0.5$ is also noticeably higher, compared to previous values, attributed to higher number of retransmissions and thus higher discrepancies across different runs. We should note that despite the considerably higher latency of the non-progressive baseline, its accuracy can be higher, since all samples – whether re-transmitted or not – exit at the final classifier. Last, we also notice a slight reduction in the average latency of SPINN's models as P_{fail} increases. This is a result of more samples early-exiting in the network, as the server becomes unavailable more often.

To sum up, these two results demonstrate that SPINN can perform sufficiently, in terms of accuracy and latency, even when the remote end remains unresponsive, by falling back to results of local exits. Compared to other systems, as the probability of failure when offloading to the server increases, there is a gradual degradation of the quality of service, instead of catastrophic unresponsiveness of the application.

5 DISCUSSION

SPINN and the current ML landscape. The status-quo deployment process of CNNs encompasses the maintenance of two models: a large, highly accurate model on the cloud and a compact, lower-accuracy one on the device. However, this approach comes with significant deployment overheads. First, from a development time perspective, the two-model approach results in two time- and resource-expensive stages. In the first stage, the large model is designed and trained requiring multiple GPU-hours. In the second stage, the large model is compressed through various techniques in order to obtain its lightweight counterpart, with the selection and tuning of the compression method being a difficult task in itself. Furthermore, typically, to gain back the accuracy loss due to compression, the lightweight model has to be fine-tuned through a number of additional training steps.

With regards to the lightweight compressed networks, SPINN is orthogonal to these techniques and hence a compressed model can be combined with SPINN to obtain further gains. Given a compressed model, our system would proceed to derive a progressive inference variant with early exits and deploy the network with a tailored implementation. For use-cases where pre-trained models are employed, SPINN can also smoothly be adopted by modifying its training scheme (Section 3.2) so that the pre-trained backbone is frozen during training and only the early exits are updated.

Nonetheless, with SPINN we also enable an alternative paradigm that alleviates the main limitations of the current practice. SPINN requires a single network design step and a single training process - which trains both the backbone network and its early exits. Upon deployment, the model's execution is adaptively tuned based on the multiple target objectives, the environmental conditions and the device and cloud load. In this manner, SPINN enables a highly customised deployment which is dynamically and efficiently adjusted to sustain its performance in mobile settings. This approach is further supported by the ML community's growing number of works on progressive networks [23, 35, 48, 72, 81, 83, 84] which can

be directly targeted by SPINN to yield an optimised deployment on mobile platforms.

Limitations and future work. Despite the challenges addressed by SPINN, our prototype system has certain limitations. First, the scheduler does not explicitly optimise for energy or memory consumption of the client. The energy consumption could be integrated as another objective in the MOO solver of the scheduler, while memory footprint could be minimised by only loading part of the model in memory and always offloading the rest. Moreover, while SPINN supports splitting at any given layer, we limit the candidate split points of each network to the outputs of ReLU layers, due to their high compressibility (Section 3.3). Although offloading could happen at sub-layer, filter-level granularity, this would impose extra overhead on the scheduler due to the significantly larger search space.

Our workflow also assumes the model to be available at both the client and server side. While cloud resources are often dedicated to specific applications, edge resources tend to present locality challenges. To handle these, we could extend SPINN to provide incremental offloading [29] and cache popular functionality [76] closer to its users. In the future, we intend to explore multi-client settings and simultaneous asynchronous inferences on a single memory copy of the model, as well as targeting regression tasks and recurrent neural networks.

6 CONCLUSION

In this paper, we present SPINN, a distributed progressive inference engine that addresses the challenge of partitioning CNN inference across device-server setups. Through a run-time scheduler that jointly tunes the early-exit policy and the partitioning scheme, the proposed system supports complex performance goals in highly dynamic environments while simultaneously guaranteeing the robust operation of the end system. By employing an efficient multi-objective optimisation approach and a CNN-specific communication optimiser, SPINN is able to deliver higher performance over the state-of-the-art systems across diverse settings, without sacrificing the overall system's accuracy and availability.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I. Venieris, and Nicholas D. Lane. 2019. EmBench: Quantifying Performance Variations of Deep Neural Networks Across Modern Commodity Devices. In *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications (EMDL)* (Seoul, Republic of Korea). 1–6.

[3] Amazon. 2020. Amazon Inferentia ML Chip. https://aws.amazon.com/machine-learning/inferentia/. [Retrieved: August 23, 2020].

[4] Alejandro Cartas, Martin Kocour, Aravindh Raman, Ilias Leontiadis, Jordi Luque, Nishanth Sastry, Jose Nuñez Martinez, Diego Perino, and Carlos Segura. 2019. A Reality Check on Inference at Mobile Networks Edge. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*. 54–59.

[5] D. Chandrasekar. 2016. *AWS Flap Detector: An Efficient Way to Detect Flapping Auto Scaling Groups on AWS Cloud.* University of Cincinnati.

[6] E. Chung et al. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[7] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[8] A. E. Eshratifar, M. S. Abrishami, and M. Pedram. 2019. JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services. *IEEE Transactions on Mobile Computing (TMC)* (2019).

[9] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 115–127.

[10] L. Fei-Fei, J. Deng, and K. Li. 2010. ImageNet: Constructing a large-scale image database. *Journal of Vision* 9, 8 (2010), 1037–1037.

[11] L. Fridman, D. E. Brown, M. Glazer, W. Angell, S. Dodd, B. Jenik, J. Terwilliger, A. Patsekin, J. Kindelsberger, L. Ding, S. Seaman, A. Mehler, A. Sipperley, A. Pettinato, B. D. Seppelt, L. Angell, B. Mehler, and B. Reimer. 2019. MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of Driver Behavior and Interaction With Automation. *IEEE Access* 7 (2019), 102021–102038.

[12] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC) (SC '18)*. IEEE Press, Article 66, 12 pages.

[13] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On Calibration of Modern Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 1321–1330.

[14] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 1 (2017),

35–47.

[15] Selim Gurun, Chandra Krintz, and Rich Wolski. 2004. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2–11.

[16] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi. 2018. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* 29, 11 (2018), 5784–5789.

[17] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).

[18] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 123–136.

[19] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629.

[20] K He, X Zhang, S Ren, and J Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[21] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 269–286.

[22] C. Hu, W. Bao, D. Wang, and F. Liu. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 1423–1431.

[23] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *International Conference on Learning Representations (ICLR)*.

[24] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *J. Mach. Learn. Res.* 18, 1 (2017), 6869–6898.

[25] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 2019. AI Benchmark: All About Deep Learning on Smartphones in 2019. In *International Conference on Computer Vision (ICCV) Workshops*.

[26] UK ISPs. 2020. 4G Mobile Network Experience Report. https://www.opensignal.com/reports/2019/04/uk/mobile-network-experience.

[27] UK ISPs. 2020. 5G Mobile Network Report. https://www.opensignal.com/2020/02/20/how-att-sprint-t-mobile-and-verizon-differ-in-their-early-5g-approach.

[28] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2704–2713.

[29] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 401–411.

[30] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the Gap Between Neural Networks and Neuromorphic Hardware with a Neural Network Compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 448–460.

[31] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-defined accelerator for large-scale DNN systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. 1–23.

[32] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 1–12.

[33] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proc. VLDB Endow.* 10, 11 (2017), 1586–1597.

[34] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 615–629.

[35] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-Deep Networks: Understanding and Mitigating Network Overthinking. In *International Conference on Machine Learning (ICML)*. 3301–3310.

[36] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 45:1–45:15.

[37] A. Kouris and C. Bouganis. 2018. Learning to Fly by MySelf: A Self-Supervised CNN-Based Approach for Autonomous Navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 1–9.

[38] A. Kouris, S. I. Venieris, and C. Bouganis. 2018. CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 155–1557.

[39] A. Kouris, S. I. Venieris, and C. Bouganis. 2020. A Throughput-Latency Co-Optimised Cascade of Convolutional Neural Network Classifiers. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1656–1661.

[40] C. Kozyrakis. 2013. Resource Efficient Computing for Warehouse-scale Datacenters. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1351–1356.

[41] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[42] V. K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley. 2018. Advanced Driver-Assistance Systems: A Path Toward Autonomous Vehicles. *IEEE Consumer Electronics Magazine* 7, 5 (2018), 18–25.

[43] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12.

[44] Stefanos Laskaridis, Stylianos I. Venieris, Hyeji Kim, and Nicholas D. Lane. 2020. HAPI: Hardware-Aware Progressive Inference. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[45] Royson Lee, Stylianos I. Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D. Lane. 2019. MobiSR: Efficient On-Device Super-Resolution Through Heterogeneous Mobile Processors. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[46] E. Li, L. Zeng, Z. Zhou, and X. Chen. 2020. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications (TWC)* (2020), 447–457.

[47] Hongshan Li, Chenghao Hu, Jingyan Jiang, Zhi Wang, Yonggang Wen, and Wenwu Zhu. 2019. JALAD: Joint Accuracy-And Latency-Aware Deep Structure Decoupling for Edge-Cloud Execution. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. 671–678.

[48] Hao Li, Hong Zhang, Xiaojuan Qi, Ruigang Yang, and Gao Huang. 2019. Improved Techniques for Training Adaptive Deep Networks. In *International Conference on Computer Vision (ICCV)*.

[49] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[50] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. 2017. MoDNN: Local distributed mobile computing system for Deep Neural Network. *Proceedings of the 2017 Design, Automation and Test in Europe (DATE)* (2017), 1396–1401.

[51] Jiachen Mao, Zhongda Yang, Wei Wen, Chunpeng Wu, Linghao Song, Kent W. Nixon, Xiang Chen, Hai Li, and Yiran Chen. 2017. MeDNN: A distributed mobile system with enhanced partition and deployment for large-scale DNNs. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2017), 751–756.

[52] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26, 6 (2004), 369–395.

[53] Szymon Migacz. 2017. 8-bit Inference with TensorRT. In *GPU Technology Conference*.

[54] Vinod Nair and Geoffrey E Hinton. 2010. Rectified Linear Units improve Restricted Boltzmann Machines. In *International Conference on Machine Learning (ICML)*. 807–814.

[55] Intel Nervana. 2020. Nervana's Early Exit Inference. https://nervanasystems.github.io/distiller/algo_earlyexit.html. [Retrieved: August 23, 2020].

[56] Miloš Nikolić, Mostafa Mahmoud, Andreas Moshovos, Yiren Zhao, and Robert Mullins. 2019. Characterizing Sources of Ineffectual Computations in Deep Learning Networks. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 165–176.

[57] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8026–8037.

[59] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. 2017. On the Expressive Power of Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, Vol. 70. 2847–2854.

[60] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 78–91.

[61] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4510–4520.

[62] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-level Deep Neural Models to FPGAs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 17:1–17:12.

[63] K. Simonyan and A. Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*.

[64] Ashish Singh and Kakali Chatterjee. 2017. Cloud security issues and challenges: A survey. *Journal of Network and Computer Applications* 79 (2017), 88–115.

[65] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 909–923.

[66] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield. 2017. Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness. In *2017 IEEE/RSJ International Conference on Intelligent Robots and*

Systems (IROS). 4241–4247.

[67] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. 2020. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[68] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI Conference on Artificial Intelligence*.

[69] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826.

[70] Willy Tarreau et al. 2012. HAProxy-the reliable, high-performance TCP/HTTP load balancer.

[71] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (LCTES 2018)*. 31–43.

[72] Surat Teerapittayanon, Bradley McDanel, and HT Kung. 2016. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2464–2469.

[73] S. Teerapittayanon, B. McDanel, and H. T. Kung. 2017. Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 328–339.

[74] Tenstorrent. 2020. Tenstorrent's Grayskull AI Chip. https://www.tenstorrent.com/technology/. [Retrieved: August 23, 2020].

[75] S. I. Venieris and C. Bouganis. 2019. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* 30, 2 (2019), 326–342.

[76] Liang Wang, Mario Almeida, Jeremy Blackburn, and Jon Crowcroft. 2016. C3PO: Computation Congestion Control (PrOactive). In *Proceedings of the 3rd ACM Conference on Information-Centric Networking (ACM-ICN '16)*. 231–236.

[77] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[78] S. Wang, A. Pathania, and T. Mitra. 2020. Neural Network Inference on Mobile SoCs. *IEEE Design Test* (2020).

[79] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*. 29:1–29:6.

[80] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344.

[81] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2246–2251.

[82] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical Evaluation of Rectified Activations in Convolutional Network. In *CoRR*.

[83] Linfeng Zhang, Jiebo Song, Anni Gao, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. 2019. Be Your Own Teacher: Improve the Performance of Convolutional Neural Networks via Self Distillation. In *IEEE International Conference on Computer Vision (ICCV)*.

[84] Linfeng Zhang, Zhanhong Tan, Jiebo Song, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. 2019. SCAN: A Scalable Neural Networks Framework Towards Compact and Efficient Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[85] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37 (2018), 2348–2359.

[86] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. In *International Conference on Learning Representations (ICLR)*.