MIRAGE: Machine Learning-based Modeling of Identical Replicas of the Jetson AGX Embedded Platform

Hazem A. Abdelhafez
Hassan Halawa
hazem@ece.ubc.ca
hhalawa@ece.ubc.ca
University of British Columbia
Vancouver, Canada

Mohamed Osama Ahmed
moahmed@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Karthik Pattabiraman
Matei Ripeanu
karthikp@ece.ubc.ca
matei@ece.ubc.ca
University of British Columbia
Vancouver, Canada

Abstract

A common feature of devices deployed at the edge today is their configurability. The NVIDIA Jetson AGX, for example, has a user-configurable frequency range larger than one order of magnitude for the CPU, the GPU, and the memory controller. Key to make effective use of this configurability is the ability to anticipate the application-level impact of a frequency configuration choice. To this end, this paper presents a novel modeling approach for predicting the runtime and power consumption for convolutional neural networks (CNNs). This modeling approach is: *(i) effective* - i.e., makes predictions with low error (models achieve an average relative error of 15.4% for runtime and 14.9% for energy); *(ii) efficient* - i.e., has a low cost to make predictions; *(iii) generic* - i.e., supports deploying updated and possibly different deep learning inference models without the need for retraining, and *(iv) practical* - i.e., requires a low training cost. Three features, all geared towards meeting the challenges of deploying in a real-world environment, set this work apart: *(i)* the focus on predicting the impact of the frequency configuration choice, *(ii)* the methodological choice to aggregate predictions at fine (i.e., kernel level) granularity which provides generality; and *(iii)* taking into account the inter-node variability among nominally identical devices.

ACM Reference Format:
Hazem A. Abdelhafez, Hassan Halawa, Mohamed Osama Ahmed, Karthik Pattabiraman, and Matei Ripeanu. 2021. MIRAGE: **M**achine Learning-based Modeling of **I**dentical **R**eplicas of the Jetson **AG**X **E**mbedded Platform. In *The Sixth ACM/IEEE Symposium on Edge Computing (SEC '21), December 14–17, 2021, San Jose, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453142.3491284

1 Introduction

Context. The compute capability deployed at the edge enables a range of applications with strict latency requirements that cannot be offloaded to the cloud. Real-time decision-making in complex environments based on computer vision in areas such as autonomous vehicles, surveillance cameras, and virtual assistants, is a particularly promising candidate.

To maximize the energy efficiency for applications running on edge devices (some of which are battery powered), recent hardware platforms employ heterogeneous processing units and shared memory supporting a wide range of operational frequencies exposed to the application developer/deployer. The promise is for more efficient energy use. The cost, however, is the increased complexity for application developers/deployers, who must explicitly choose which processing units to target and how to configure them to meet the application's Quality of Service (QoS) objectives, while minimizing energy consumption.

Objective. Our long-term objective is to enable automating this process for applications employing ML inference at the edge. As a first step in this direction, the goal of this study is to explore the feasibility of a prediction engine (for performance and power consumption) that can be later leveraged to automate the tuning of the underlying platform to achieve high energy efficiency while meeting the target QoS objectives. This prediction engine should be: *(i) effective* - i.e., predictions have low error; *(ii) efficient* - i.e., predictions have a low (runtime) cost thus enabling online adaptation for dynamic environments/workloads via automated platform tuning; *(iii) generic* - i.e., deploying updated and possibly different deep learning inference models, or using different type of input data (e.g., changing image resolution) should not require model retraining, and *(iv) practical* - i.e., the training cost is not onerous.

Workload. We focus on one workload category commonly deployed on edge devices: convolutional neural networks (CNNs) in computer vision applications - often used in contexts that have soft real-time constraints and stringent energy budgets.

This workload choice was motivated by three factors: *(i)* CNNs (and AI in general) are at the core of many Edge applications (e.g., video surveillance, home automation, remote healthcare) which would benefit from the ability to make performance and power consumption predictions, *(ii)* several industry-backed benchmarks (e.g., MLPerf [42], EdgeBench [14], AIMatrix [57], MLMark [50], EdgeAIBench [21]) focus on CNNs and other ML models to benchmark edge platforms, and *(iii)* CNN kernels (e.g., matrix multiply, convolution) are at the core of several AI/ML applications.

Challenges. Predicting the impact of frequency configuration selection on runtime and power consumption raises several challenges:

- *The need to quantify the impact of the frequency configuration choice in a large configuration space*, where the choice of the

SEC '21, December 14–17, 2021, San Jose, CA, USA

ACM ISBN 978-1-4503-8390-5/21/12…$15.00
https://doi.org/10.1145/3453142.3491284

configuration has a sizeable, application-dependent, and non-linear impact on the performance and power consumption of an application (§2.1).
- *The diversity and frequent evolution of the deployed networks* as a result of: *(i)* additional labeled data available for training, and/or *(ii)* continuous advances in the deep learning field that yield increasingly robust and accurate networks. As a result, over the lifetime of a deployed application, models are continuously re-trained then re-deployed at the edge, and, in some cases, models are replaced by completely different ones (architecture wise).
- *The variability, in performance and power consumption, between nominally 'identical replicas' of edge devices (i.e., devices of the same hardware model and running the same software stack)* (§3.1) can significantly degrade the prediction quality if not taken into account during training.

Approach. We summarize our machine learning-based approach and highlight its distinguishing features:
- *Kernel-level decomposition.* Regardless of their specific architecture, CNNs incorporate the same essential building blocks (*kernels* [30]: e.g., convolution, pooling, batch normalization, matrix multiply), but differ in how these are used. We explore whether modeling the performance and power consumption at the kernel level enables generality: that is, whether predictions at the kernel-level *(i)* can be made accurately, and *(ii)* can be aggregated to obtain accurate network-level predictions for arbitrary CNNs.
- *Kernels extraction.* We characterize ten of the most popular CNNs (e.g., VGG [45], GoogLeNet [46], SqueezeNet [24]) to extract the kernels which meet two criteria: *(i) significance* – they collectively constitute the majority of the runtime of an inference task, and *(ii) commonality* – they are frequently used across different networks. Our findings (§3.2) are in line with previous studies [4] and represent the first quantitative analysis of these CNNs on the Jetson AGX platform.
- *Training data generation.* To collect training data we run the selected kernels against different Dynamic Voltage and Frequency Scaling (DVFS) configurations and measure their runtime and power consumption. One of the challenges is the large number of input parameters - with a wide and continuous range of values - for these kernels. For example, the 2D convolution kernel requires nine parameters (e.g., weight shape, padding, dilation, etc.) that specify the behavior of the kernel, besides the input data shape. To overcome this challenge we propose a space-reduction technique (§4.1) that combines random sampling and domain-specific knowledge.
- *Model training.* With this data, we train regression models using XGBoost [12] and the BOHB [16] optimizer for hyperparameter tuning (§4.3). A distinguishing feature of our approach is that we collect training data from *multiple* nominally identical AGX boards. This is for two reasons: *(i)* to expose the model to the inter-node variability (discussed in §3.1), and *(ii)* to accelerate the process of generating training data in a large space (DVFS, multiple kernels, kernel parameters, and input shapes).

Contributions. The main contributions are as follows:
- *Workload characterization.* We characterize the performance of 10 widely used computer vision CNNs on the Jetson AGX (§3.2). We find that, on average, the Convolution, Matrix Multiply, and Batch Normalization kernels offloaded to the GPU collectively account for the majority (>90%) of the inference task runtime for most of these CNNs. We also find that the runtime of eight kernels combined represents on average ≈96% of the networks' inference time.
- *A generic approach to predicting runtime and power/energy consumption for computer vision inference networks.* We create kernel-level machine learning models that can be used to estimate the runtime and average power consumption for a wide range of computer vision CNNs (§4). The fact that our approach works at a kernel level granularity makes it *generic* as it allows predicting the runtime and energy consumption of an inference task at the network level for arbitrary CNNs not seen during training.
- *An evaluation of these models.* We evaluate the accuracy of these models both at the kernel level (§6.1) and at the network level (§6.2). The kernel-level evaluation (§6.1) shows that the kernel-level models have low prediction error (e.g., on average 73.92% and 55.59% of the runtime and power models respectively have relative error of less than 5%). The network-level evaluation (§6.2) shows that the network level models are effective (e.g., predictions have, on average, a 15.4% error for runtime, and 14.9% for energy). We also evaluate the runtime overhead of the models (§6.4) and find that they are efficient: i.e., they have low prediction time overhead compared to the corresponding kernel runtime. Additionally, the models are practical: the median training time across all the models is 4.9 hours for runtime (1.4 hours for power). Finally, we quantify the 'cost of generality' (i.e., modeling at the kernel level). The results (§6.3) show a low degradation in prediction quality of the kernel-level approach compared to a network-level approach (on average, the prediction error increases by 11.76% for runtime and 10.01% for energy).
- *We demonstrate that taking inter-node variability into account improves the quality of the prediction models (§6.4).* We extend a previous characterization study to strengthen the argument that inter-node variability is significant for both runtime and power (§3.1). This motivates us to account for inter-node variability during model development (§4). We show that our approach: *(i)* improves the quality of predictions (e.g., for runtime and power the root mean square error (RMSE) is improved on average by 9.5% and 13.6%); and *(ii)* reduces prediction error variability compared to a traditional model training approach not taking inter-node variability into account (§6.4).
- *We describe our experience training performance/power models as well as lessons learned about the Jetson AGX platform.* These are summarized in §7 and discussed throughout this paper.

2 Background

This section presents two key pieces of background information: *(i)* it highlights the large and consequential frequency configuration space of the NVIDIA Jetson AGX boards, and *(ii)* it presents background information on convolutional neural networks (CNNs).

2.1 Large frequency configuration space.

The Jetson AGX board is highly configurable over multiple axes. Key is the ability to configure the operational frequency over a range spanning over one order magnitude for the CPU (0.1–2.7GHz), GPU

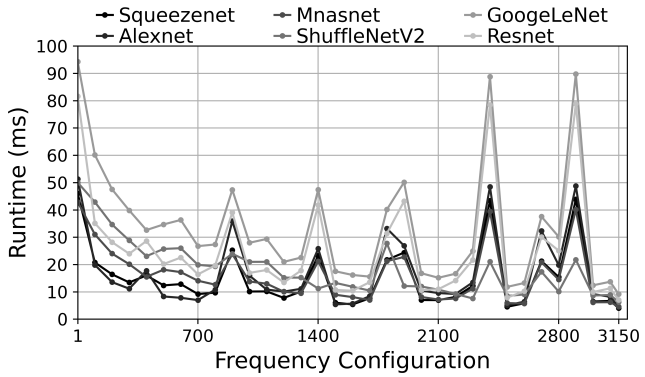


Figure 1: Frequency configuration impact on runtime for different CNNs. The y-axis shows the inference runtime (ms) and the x-axis shows the configuration identifier. The configurations are sorted in ascending order using CPU, GPU, and memory frequencies as sorting keys, then the runtime for each 100^{th} configuration is plotted. The fact that the x-axis linearizes the three dimensional frequency configuration space explains the 'spikes' visible in the plot.

(0.1–1.4GHz), and the memory controller (0.2–2.1GHz) in discrete steps. This leads to a large configuration space with $\approx 3.6K$ unique combinations (i.e., *frequency configurations*) each with different performance and power characteristics. Other edge platforms offer similar configurability features (e.g., Raspberry Pi), however, what differentiates the Jetson AGX board (as well as other boards from the NVIDIA Jetson line) is the large $10-20\times$ dynamic frequency range (e.g., compared to an up to $3\times$ range for most Intel processors). We note that making good choices in the frequency configuration space is: *(i) important*. This is highlighted by the large application-level performance (and power) impact of the configuration choice (over $10\times$); and *(ii) non-trivial*. This is highlighted by the fact that this space not 'smooth' and that frequency configuration changes have different impact on different applications. Fig. 1 presents visually some of these observations.

2.2 CNNs: structure and kernels.

Convolutional neural networks (CNNs) [28] are a class of deep neural networks (DNNs) specialized for multimedia (e.g., image [27], video [44], and speech [2]). CNNs are used for image/video analysis (e.g., object detection [10], recognition [27], semantic segmentation [39]) where they have proven to be versatile and effective. They also represent the core building blocks of several edge benchmarks (e.g., MLPerf [42], EdgeBench [14], MLMark [50]).

A DNN's main building block is a feed-forward neural layer. Typically, a DNN consists of a single input and a single output layer in addition to one or more hidden layers in-between. Each layer consists of multiple neurons. A neuron is modeled as a linear mathematical function that multiplies its inputs (X_i) by certain weights (W_i), and adds biases (B_i). To model non-linear systems, a non-linear activation function f is applied on the neuron's output. A single neuron equation can be represented as: $Y_i = f(X_i * W_i + B_i)$.

The most common type of layer in DNNs is a fully-connected layer in which each neuron is connected to all the neurons from the previous layer. The word 'deep' in DNN refers to the number of hidden layers. Adding more layers can improve network accuracy but it increases the complexity of the learning process dramatically.

CNNs take a different approach by reorganizing the hidden layers as 2D filters. These filters typically have small dimensions (e.g., 3×3) compared to the input image, and are applied across the input image (i.e., left to right and top to bottom); thus, extracting feature maps for each patch of the input image. Applying these 2D filters to patches of the input image involves element wise multiplication between the filters' weights and input pixels' values (i.e., convolution). In traditional computer vision systems, convolution filters' weights were hand-crafted by experts to extract certain distinctive features (e.g., edges, corners, texture). In modern CNNs the network learns the weights of the convolution filters during the training process without the need for domain-specific expertise.

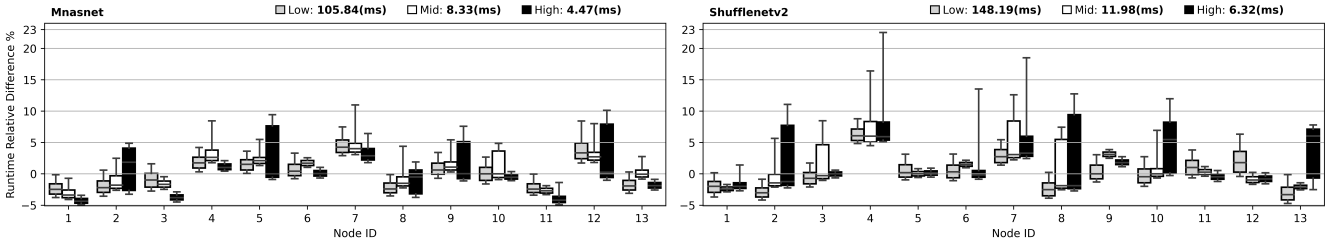
Generally, a simple CNN incorporates a permutation of n feed-forward layers. The dominant types of layers in CNNs are:

- *Convolution layer* – the main building block for CNNs, where most of the computations occur. In this layer, a *convolution* kernel is applied to the input image to extract certain features. Then a non-linear activation function, e.g., a Rectified Linear Unit *ReLU*, is applied to the output of the convolution operation to introduce non-linearity to the model.
- *Pooling layer* to: *(i)* reduce the space of the input feature map from the convolution layer, *(ii)* extract the dominant feature from the feature map, and *(iii)* handle translation invariance and noise. There are several kernels to apply pooling, such as: *MaxPool* (gets the maximum value from a 2D matrix), *AveragePool* (calculates the average value of all elements of a 2D matrix), and *AdaptiveAvgPool* (similar to AveragePool but with variable padding and stride size applied to the 2D input to maintain a user specified output size).
- *Fully-connected layer* used after extracting features using the convolution and pooling layers to flatten the output to a 1D feature vector. This vector is now a suitable input for a traditional fully-connected layer of neurons that learns to classify the different input feature vectors to their corresponding classes based on the training data. In the fully-connected layer, the input feature vector is multiplied using the matrix multiply (*MatMul*) kernel by the learned weights, and the offset is added using the *Add* kernel before applying a *SoftMax* activation function to classify the feature vector to a certain class.

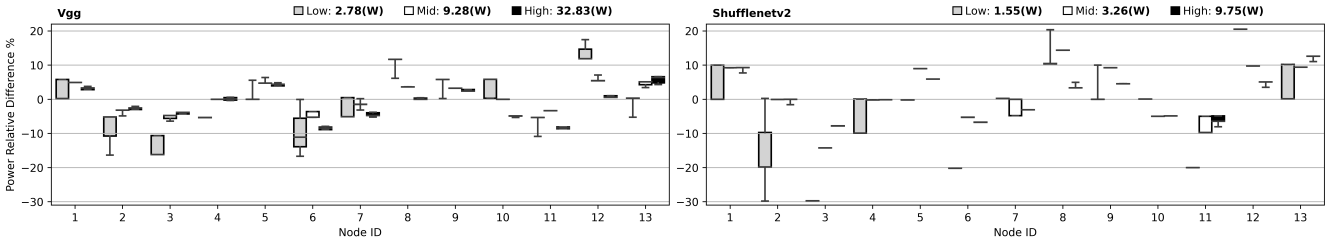
We focus on ten of the most popular CNNs (Squeezenet, AlexNet, MobileNetv2, MNASNet, ResNet, ShuffleNetv2, GoogLeNet, Inception3, VGG, and DenseNet) as implemented by the Torchvision [33] library (details in §5.2).

3 Characterization Studies

This section presents two characterization studies that motivate the decisions we make when developing our methodology: *(i)* a variability characterization which highlights the existence and the magnitude of variability amongst *nominally identical* Jetson AGX boards, and *(ii)* a workload characterization which uncovers the main building blocks (i.e., kernels) of frequently used CNNs.



(a) Relative difference percentage *RD* for the runtime.



(b) Relative difference percentage *RD* for the power consumption.

Figure 2: Relative difference percentage *RD* values for runtime and power consumption for two pairs of CNNs that manifest mid (left) and highest (right) runtime inter-node variability (respectively, lowest and highest for power). The x-axis shows the node ID (AGX board identifier), and the y-axis indicates the relative difference percentage between observations and the median of medians. The three boxplots per node represent the lowest, midst, and highest frequency configurations (in gray, white, and black respectively). The legend on top of each plot shows the absolute value for the median of medians for each configuration. For each boxplot, the top and bottom sides represent the first and third quartiles (Q1 and Q3). The horizontal line represents the median (Q2). The top/bottom fences indicate the top/bottom deciles (i.e., 10% and 90% thresholds of the distribution).

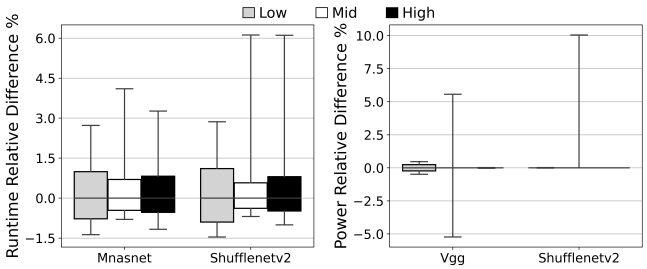


Figure 3: Intra-node variability characterization for 13 nodes, depicted using the boxplot of the relative difference percentage *RD* (y-axis) between multiple measurements, and the median value for different CNNs (x-axis) on a per-board basis. The boxplots present the same thresholds as in the previous figure.

3.1 Variability characterization.[1]

Background. We define variability as a significant difference between measurements of metrics of interest (e.g., runtime and power consumption) under the same constraints (e.g., workload, software stack, hardware platform). Variability can be of two types:

(i) *intra-node* - i.e., variability over time for an application deployed on a single board (i.e., node). Possible sources are: *(i)* interference with operating system services or other applications, *(ii)* initialization differences, and/or *(iii)* ambient context [19, 55] (e.g., temperature leading to CPU throttling).

(ii) *inter-node* - i.e., variability across several identical (i.e., same software and hardware) boards. Possible sources are: *(i)* manufacturing process variation and/or *(ii)* hardware components' aging [20, 34].

While the existence of variability is well-known [19, 55], with the exception of Large-scale and High-Performance Computing (HPC) environments [15, 26, 54], *inter-node* variability has often been ignored at the edge as it is assumed to be negligible with no significant impact on the target applications.

Objective. We aim to: *(i)* quantify the inter-node variability among several nominally identical Jetson AGX boards; and *(ii)* compare inter- and intra- node variability to highlight their significance from an application perspective.

Terminology. We define a *frequency configuration* as a unique combination of CPU, GPU and memory controller frequencies. For each frequency configuration, each workload, and each board, we collect a *measurement sample* as a set of ***N*** *observations.*

Methodology. We collect runtime and power consumption *measurement samples* for the inference task for 10 CNNs (§3.2) on 13 *nominally identical* Jetson AGX boards (§5) for about 500 frequency configurations chosen to uniformly sample the entire configuration space. After discarding warm-up runs, each *measurement sample* has 10,000 observations for both runtime and power. We take all precautions to eliminate the impact of other factors (e.g., software stack version, application interference, temperature, fan) on our measurements (§5.2).

Results. We have confirmed [3] using statistical tests (K-samples

[1]This subsection summarizes and extends a previous workshop publication [3]. Compared with this workshop publication, this subsection: *(i)* provides increased confidence - 100× larger measurement samples; *(ii)* compares to intra-node variability; and *(iii)* uses more CNNs.

Anderson Darling [43] and Two-samples Kolmogorov-Smirnov [23]) that, for a majority of the configurations, the boards are statistically different (i.e., runtime and power observations for different boards come, with high confidence, from different distributions). Rather than presenting a summary of these tests, here we focus on a visual presentation as we find it more effective to convey the presence of inter-node variability and indicate its magnitude.

We focus on three *frequency configurations* representing the low, middle, and high points of the frequency range. For each *frequency configuration* and workload, we group the data per board and calculate the median value of the corresponding *measurement sample*. Then we calculate the median of medians to have a *reference value* for each metric we consider. Finally, for each sample, we plot the relative difference percentage (*RD*) between each observation in the *measurement sample* and the *reference value* as boxplots.

Due to space limitations, we only plot two CNNs for each of runtime and power. For runtime, we rank CNNs by the observed inter-node variability and Fig. 2a presents the inter-node variability for MnasNet (mid-range) and ShufflenetV2 (highest variability). For power, inter-node variability is visible for all CNNs: Fig. 2b presents the extremes: VGG (lowest) to ShuffleNetV2 (highest variability).

The plots highlight that the runtime/power observations for different boards are (likely) drawn from different distributions. This is evident for power (Fig. 2b), where even the median of the observations vary among the 13 boards between about 20% (for VGG) to over 30% (for ShuffleNetV2) of the reference value. For runtime (Fig. 2a) the differences are less pronounced but still visible with the naked eye: focus on ShuffleNetV2's runtime for example; here the variation among boards is up to 10% of the reference value for the top quartile threshold, and up to 25% for the top decile threshold.

Finally, Fig. 3 highlights that intra-node variability is much lower. Each boxplot in Fig. 3 aggregates 130k observations for a CNN at one frequency configuration point. The *reference value* is the median of the observations of the *measurement sample* of each node.

The key takeaways from this variability characterization study are: *(i)* inter-node variability on the Jetson AGX is significant (and particularly large for power consumption), hence it must be accounted for when building a generic prediction engine; *(ii)* inter-node variability is significantly larger than intra-node variability; and *(iii)* the magnitude of the variability is workload dependent (i.e., it changes depending on the target CNN), which, we speculate, is a result of each CNN stressing different hardware components (e.g., CPU, GPU).

Lastly, while we characterize variability on the Jetson AGX and highlight its impact as a challenge in realistic edge deployment scenarios, a logical next step of this characterization is root-cause analysis. Hence, as part of our future work, we plan to extend the characterization study to investigate different factors that might be contributing to this variability (e.g., workload characteristics, state initialization effect, certain frequency ranges, etc.).

3.2 Workload characterization.

Objectives. Our primary goal is to characterize the workload generated by CNNs, that is, to quantify the runtime contribution of the underlying computational kernels to the end-to-end network inference task. Our secondary goal is to understand whether there exists a small group of kernels that are common across all CNNs and generate most of the load.

Methodology. We place custom probes in PyTorch's [40] source code to extract the kernels that each CNN launches, along with their parameters (e.g., convolution kernel stride and padding sizes) and input shapes (see §5 for details). We tune the 13 Jetson AGX boards to the maximum *frequency configuration*, and collect the total runtime for each kernel launched by each CNN (as a *measurement sample* of 100 observations per board). Then, we calculate the median (which is less susceptible to outliers compared to the mean) per board, and finally, we calculate the median of medians (across all boards). The result is used as the absolute runtime contribution of each kernel T_{Ki} to the network's inference time. Finally, we collect a runtime *measurement sample* for each CNN on each board, and similar to the kernels, we calculate the median of medians across all boards to represent the network's inference time T.

Kernel selection is carried out in two phases: the first one picks the kernels that collectively represent a large majority ($\geq$ 95%) of the runtime for each network, and the second one picks the kernels that appear in at least 40% of the studied networks. The first phase prioritizes the runtime contribution of the kernels (i.e., significance). The second phase makes sure that these kernels are common. As such, they are useful for other vision-based ML applications and not specific to the networks we analyzed (i.e., commonality).

Results. The number of unique kernels that are run by each network varies from 7 (SqueezeNet) to 16 (InceptionV3), while the number of kernel launches varies from 28 (AlexNet) to 580 (DenseNet). Table 1 details the runtime contribution of each kernel. The *2D convolution* kernel is called in all networks and it has the largest mean value (last row) of the runtime contribution percentage, followed by the *2D batch normalization* and *matrix multiply*. The percentages highlight the significance of each kernel and are broadly in line with similar studies on other platforms [4, 30].

T_k is the runtime summation of the *selected kernels* (i.e., the most common and significant ones), and $T_{k'}$ is the runtime summation for the left-out kernels (i.e., the kernels that have a minor contribution to each network's end-to-end inference time). T represents the end-to-end network inference time as measured from the high-level code (i.e., the $forward()$ method) in the PyTorch framework.

The $\frac{T_k}{T}$ column highlights that the *selected kernels* represent, on average, ≈96% of the inference time. With the exception of ShuffleNetV2, the minimum value for $\frac{T_k}{T}$ is 97.43%. [2] Percentages tend to be greater than 100% due to the overhead associated with timing each kernel individually versus timing the network inference time as a single black box.

Note that among the left out kernels, not included in Table 1, is the *FusionGroup* kernel. FusionGroups are CUDA kernels that replace one or more of the target network's computation graph nodes (e.g., a node represents an operation such as Conv2D, or MatMul). They are created during the GraphFuse optimization pass within the PyTorch Just In Time (JIT) compilation step (see §5.2 for more details). In 70% of the networks we studied, the FusionGroup replaces a single call of the 2D batch normalization kernel. Hence, $T_{K'}$ is representative of the left-out kernels even without these.

[2] By inspecting the NVIDIA Profiler tool [37], we found that all the GPU compute kernels, and data transfer operations consume ≈43% of ShuffleNetV2's inference time, and we believe that the rest of the inference time is spent on the CPU.

Table 1: The runtime contribution and breakdown of the selected and left-out kernels. Each row corresponds to a specific network. The columns from 2 to 9 show a kernel's runtime percentage of the total network inference time. Cells with a '-' value indicate that the kernel was not called in the respective network. T_K and $T_{k'}$ are the summations of the runtimes for the selected and left-out kernels respectively. T is the network's inference time (measured end-to-end). The last row shows the mean across all networks. The kernels (columns 2 to 9) are sorted based on the mean values (last row) of their contribution percentage to the networks' inference time.

Network	*Selected kernels contribution to inference time (%)*								*All kernels runtime breakdown (ms and %)*					
	Conv2D	Batch Norm2D	Matrix Multiply	Relu	Cat	Max Pool2D	Add	Adaptive Pool2D	$T_K(ms)$	$T_{K'}(ms)$	$T(ms)$	$\frac{T_K}{T}(\%)$	$\frac{T_K}{(T_K+T_{K'})}(\%)$	$\frac{T_K+T_{K'}}{T}(\%)$
Alexnet	45.26	-	55.88	1.68	-	1.18	0.36	0.49	4.76	0.004	4.54	104.81	99.92	104.89
Densenet	84.64	9.38	0.17	5.28	5.32	0.16	0.01	0.04	54.68	0.139	52.09	104.98	99.75	105.25
GoogeLeNet	78.06	6.75	0.48	5.29	2.22	4.57	0.05	0.12	9.23	0.009	9.46	97.54	99.90	97.64
Inception3	89.93	6.55	0.33	4.18	1.61	0.87	0.19	0.07	26.44	0.895	25.50	103.66	96.73	107.17
Mnasnet	66.77	22.44	1.24	11.00	-	-	1.91	-	4.63	0.020	4.47	103.41	99.57	103.86
MobileNetV2	66.67	27.29	1.28	-	-	-	1.85	0.31	4.16	0.587	4.27	97.43	87.64	111.17
Resnet	95.99	4.83	0.38	3.37	-	0.83	1.72	0.12	7.51	0.006	7.00	107.25	99.91	107.34
ShuffleNetV2	24.61	8.11	0.72	4.57	4.06	0.40	0.07	-	2.69	0.066	6.32	42.53	97.60	43.58
Squeezenet	78.00	-	-	10.36	6.52	4.31	-	0.42	4.04	0.001	4.06	99.58	99.97	99.61
VGG	83.61	-	15.09	2.58	-	1.09	0.04	0.10	37.58	0.004	36.62	102.60	99.99	102.61
Mean	**71.35%**	**12.19%**	**8.40%**	**5.37%**	**3.95%**	**1.68%**	**0.69%**	**0.21%**	**15.57**	**0.173**	**15.43**	**96.38**	**98.10**	**98.31**

A key takeaway from the workload characterization study is that eight kernels (convolution, batch normalization, matrix multiply, relu, cat, max pool, add, and adaptive average pool) represent on average ≈96% of the CNNs runtime on the Jetson AGX platform.

4 Modeling Methodology

While the main components of our methodology are largely mainstream, compared to prior work (§8), our runtime/power modeling methodology puts forward three innovations: *(i)* working at a kernel-level granularity, which makes the developed models *generic* in the sense that new CNNs can be modeled without retraining; *(ii)* taking into account the existing inter-node variability (§3.1) which increases the prediction quality of the developed models (§6.4; and *(iii)* the space sampling methodology to efficiently generate test data (§4.2).

4.1 CNNs' analysis and kernel extraction.

A key first step is to select the most significant and common kernels. §3.2, discusses our methodology to identify the kernels that meet these criteria. The main intuition is that if we model the *selected kernels*, we can aggregate their runtime and power/energy consumption predictions to obtain network-level estimates.

4.2 Training data generation / pre-processing.

Each compute kernel (e.g., MatMul, Conv2D, ReLU) has three relevant features: *(i)* input characteristics - the shapes of the tensors usually manipulated by the kernel, and the precision of the individual tensor elements, e.g., FP16 or INT8; *(ii)* kernel parameters that specify the behavior of the kernels (e.g., stride size for the convolution kernel), and *(iii)* implementation algorithms. For example, in PyTorch the 2D convolution kernel has nine parameters. Depending on the values of these parameters and the input shape, the underlying cuDNN [13] library chooses a specific implementation/algorithm [30]. Hence, the complexity of modeling these kernels becomes a challenge given that we need to account for the continuous ranges for the input shapes and kernel parameters, and for several underlying implementations. To overcome this challenge we: *(i)* create implementation-agnostic models based only on the input characteristics and kernel parameters, and *(ii)* reduce the space for the inputs and kernel parameters during model training.

We designed a domain-specific space reduction technique for the inputs and kernel parameters with the goal of decreasing the training time, and improving the overall prediction quality. For all kernels, we summarize the ranges of input shapes, and kernel parameters seen for all of the networks listed in Table 1 based on profiling during the characterization experiments. We then create a range that covers all values obtained during profiling, which gets sampled randomly when generating training and testing data. For example, assuming that in the 10 networks the input matrices of the MatMul kernel have a number of columns that mostly fall in the range $[1K, 4K]$, then we randomly pick values from $[1K-e, 4K+e']$, where e, e' are margin values that slightly extend the range beyond the minimum and maximum values. This range extension exposes the learning algorithm to data points that might be seen in similar networks but do not appear in our CNN characterization study.

There is a trade-off between the training time and the e, e' margin values. This trade-off impacts the generality and accuracy of the trained models. On the one side, choosing a larger range increases the generality of the model and the chance that the model will perform well on data points not seen during profiling. On the other side, it requires expanding the training data size (leading to longer training time) to make sure that the space is sampled sufficiently for the model's accuracy to be reasonably high. Finally, any outliers (e.g., 32 and 25K columns in the MatMul) are added to the space as additional ranges $[25k - e, 25k + e']$, and $[32 - e, 32 + e']$ (instead of a single extended range $[32 - e, 25k + e']$). The input tensors are filled with random FP32 numbers from the standard normal distribution (i.e., mean=0, variance=1).

The only data pre-processing steps we apply to the training and testing data are removing redundant/dependent features (e.g., first input matrix columns and second input matrix rows in MatMul kernel), and log-transforming the target label (runtime and average power) to avoid negative-valued predictions.

For the convolution kernel, there are nine different parameters, which reduces the probability of randomly generating the combinations that are seen in real-world networks. We use the weighted-sampling technique used in imbalanced learning problems [22] to sample the training space. This technique either picks a random

combination of kernel parameters or a real-world one (i.e., seen in any of the CNNs we study) with a 50% chance each.

Discussion. On the one side, our approach enables: *(i)* building a practical model as there is no dependency on extra tools, such as the NVIDIA profiler, to extract the underlying CUDA kernels, and *(ii)* easily being extended to other frameworks that use similar high-level APIs (e.g., TensorFlow [1]). Finally, limiting the value space for the input shapes and kernel parameters based on data extracted from real-world networks and workloads exposes the trained models to representative data and yields accurate predictions.

On the other side, this approach has two potential drawbacks: *(i)* adopting an implementation agnostic path discards features that might be useful when modeling the kernels, that is, which algorithm is going to run given a certain input and kernel parameters. (Exposing this feature, however, requires deep expert knowledge with the libraries or low-level profiling using the NVIDIA tools), and *(ii)* reducing the space for the inputs and kernel parameters limits the model's exposure to a wider range of values, and limit its generality if non-representative values are used during training.

4.3 Model training and tuning.

Background. We use XGBoost [12] to build the runtime and average power consumption models. XGBoost is a scalable and portable implementation of gradient tree boosting [17], a widely used technique in regression and classification problems [12]. Boosting is an ensemble-based technique that constructs a strong learner by aggregating multiple weak learners (e.g., regression trees) to achieve better performance based on a certain loss function [18, 25, 35, 41]. A gradient descent algorithm is used to minimize the loss before adding a new weak learner.

XGBoost encompasses several hyperparameters that influence a trained model's performance. There are several statistical methods to tune hyperparameters such as grid search, random search, and Bayesian optimization. We use a variant of Bayesian Optimization with HyperBand search (BOHB) [16, 29]. Bayesian optimization (BO) builds a probabilistic model of an objective function using the observed points. Hyperband uses the notion of a budget to evaluate the objective function. As the budget increases, the training time increases. Examples of the budget include the number of: epochs, training samples, and estimators. Hyperband evaluates the objective function on small budgets first, to eliminate non-promising configurations early, then the best configurations are evaluated on larger budgets.

Training procedure. To address inter-node variability (discussed in §3.1) and to accelerate data collection, we collect training and testing data from 13 nominally identical Jetson AGX boards for each of the eight kernels we select for modeling. We split the 13 boards into two groups: *(i)* training (11 boards), *(ii)* testing (2 boards).

For each kernel, we collect a different number of data points (which differ in the frequency configuration used as well as the kernel inputs). The number of data points we collect for each kernel is directly proportional to two factors: *(i)* the significance of the kernel - to enhance the accuracy of the kernels that make up most of the networks' inference task runtime and energy consumption, and *(ii)* the number of parameters the kernel has - to expose the models to more data ranges. For example, for the convolution kernel runtime model (that has nine input parameters), we collect ≈1,400,000 data records, while for the matrix multiply runtime model (that has two input parameters), we collect ≈150,000 data records only.

We employ *leave-one-out cross-validation* to tune the models' hyperparameters. In this technique, the input training data is split into two groups: *(i)* training (ten boards), and *(ii)* validation (one board). We further change boards 10 times per hyperparameter tuning iteration to cover all boards, so that each board is chosen as a validation board exactly once. The evaluation results of the cross-validation steps are averaged to summarize the quality of the hyperparameters being evaluated at the current hyperparameter tuning iteration. This average value drives the BOHB hyperparameter tuning algorithm to find the best fitting set of hyperparameters.

There are several hyperparameters that highly influence the accuracy and complexity of XGBoost regression and classification models. Table 2 reports the hyperparameters we tune using BOHB and the ranges from which their values are chosen. This table can be used to reproduce the models developed in this paper, and it indicates the large size of the hyperparameter tuning space.

Table 2: A brief summary of the XGBoost hyperparameters tuned during the training process [56].

Parameter(s)	Type	Range
n_estimators	Integer	[100, 3000]
max_depth	Integer	[5, 11]
min_child_weight	Integer	[5, 8]
subsample/colsample_bytree	Float	[0.8, 1]
learning_rate	Float (log scale)	[0.005, 0.1]
reg_alpha/reg_lamda	Float (log scale)	[0.0001, 0.01]

Table 3: NVIDIA Jetson AGX specifications as reported by the OS and collected from the relevant documentation [47].

	AGX Platform
CPU	8-core ARM v8.2 64-bit CPU
Architecture	ARMv8-A
L1d/L1i/L2/L3 Cache	64KB/128KB/2MB/4MB
Min/Max Core Frequency Range	**115.2MHz → 2.265GHz**
Peak Theoretical FLOPS	144.96 GFLOPS
GPU	Volta: 4 TPCs | 8 SMs | 512 CUDA Cores | 64 Tensor Cores
L1/L2 Cache	128KB/512KB
Frequency Range	**114.75MHz → 1.377GHz**
Peak Theoretical FLOPS	1.4 TFLOPS
DRAM	16GB LPDDR4x (2133MHz, 2 Channels)
Data Bus Width	256bit
Min/Max EMC Frequency Range	**204MHz → 2133MHz**
Peak Theoretical Bandwidth	136.512 GB/sec

5 Experimental Setup Details

5.1 Hardware.

The Jetson embedded computing family combines NVIDIA's GPUs with low-powered ARM CPU cores in a shared-memory architecture. NVIDIA regularly updates their Jetson platform to include their most advanced GPU and ARM cores as well as to add new processing elements (e.g., Deep Learning and Programmable Vision Accelerators). We focus on the most powerful Jetson platform, the AGX. Table 3 summarizes its technical specifications. We built a cluster of 13 Jetson AGX boards (same vendor model/SKU).

The CPU, GPU, and memory controller can operate at 29, 14, and 9 frequency scaling levels (see Table 3). This brings the frequency configuration space to ≈3.6K combinations.

5.2 Software

We install the latest NVIDIA JetPack SDK version 4.4 on all boards. The SDK incorporates the latest Linux OS for Tegra and driver package (L4T v32.4.6) for the Jetson platform. Moreover, it has a full set of optimized software libraries (e.g., CUDA v10.2) to build applications targeting the various on-board processing elements (e.g., GPU, deep learning and vision accelerators).

Machine learning: PyTorch Framework. We use PyTorch [40] (v1.6) one of the most popular deep learning frameworks. The core functionality is implemented in C++ and exposed as Python C++ extensions to the users (i.e., LibTorch).

PyTorch has two operational modes: *(i)* Eager Execution which is suitable for research, debugging, and prototyping, and *(ii)* Scripting well-suited for performance-sensitive scenarios (e.g., production deployments). In the Scripting mode, PyTorch code is translated to TorchScript - a statically-typed subset of the Python language and intermediate representation (IR) - via the PyTorch just-in-time (JIT) compiler. The JIT first converts the PyTorch code to a graph representation, and then applies several optimizations (e.g., Fusing operations, eliminating dead code) before emitting the optimized TorchScript IR. The generated IR is decoupled from Python - avoiding some of its performance disadvantages such as multi-threading performance issues, and the global interpreter lock (GIL) - and can be run from C++ for performance sensitive deployments. To execute a TorchScript IR, the JIT module compiles the IR to a bytecode representation, which is run in a stack-based Virtual Machine (VM).

Our profiling in TorchScript. We placed custom probes in the VM's interpreter that executes the TorchScript IR code to collect the information for our modeling. There are two reasons why we decided to place these probes in the interpreter: *(i)* it allows us to capture the actual operations run by the interpreter post-optimizations, so we do not have to worry about internal optimizations (e.g., fusing operations) or decomposition (e.g., for the Linear layer to separate the MatMul and Add tensor operations) that might occur; and *(ii)* it provides a single point at which we can extract the operations (e.g., Convolution, MatMul), their kernel parameters and input characteristics (e.g., count and shape) regardless of the target network.

Machine learning: Pre-trained CNNs. We use Torchvision [33] version 0.8, to load the pretrained CNNs used in the experiments. Torchvision is a popular machine-vision package - compatible with and part of the PyTorch project - that encompasses popular data sets, models, and image transformations to process visual data. All classification CNNs included in Torchvision are trained and optimized on the 1000-class ImageNet dataset. Each vision network accepts inputs of a specific size (i.e., 4D Tensor). We generate shape-compatible input tensors (filled with randomly generated FP32 numbers in the range [0, 1]) for each network to be used as an inference task's input. This imitates 3-channel input images typically used in the classification tasks of these CNNs. We study ten networks: AlexNet, GoogLeNet, ResNet, MobilenetV2, MNASNet, SqueezeNet, ShuffleNetV2, DenseNet161, VGG16, and InceptionV3.

5.3 Data Collection: Runtime and Power

Timing measurements. We use the CUDA Events system [48] for measuring the networks' inference time (0.5µs resolution). We discard the first few warm-up iterations before performing our analysis.

Power measurements. For accurate power measurements, we leverage the two on-board INA3221 [49] (0.5% error) Power Monitoring Units (PMU) that can be read via an exposed virtual file system (i.e., sysfs). PMUs have six power 'rails' (for CPU, GPU, memory module, computer vision (CV) accelerator, auxiliary on-chip components, system IO) which measure the power each component draws. We discard the CV accelerator and system IO rails as they are not relevant to our experiments.

Size of measurement samples. For each measurement we collect multiple observations. For runtime, each *measurement sample* contains 105 *observations* (the first five are discarded as warm up). For power, each *measurement sample* contains 60 observations, and we discard the first 10. The median of each *measurement sample* is then used for modeling and evaluation.

5.4 Controlling the environment

Eliminating other sources of variability. We create a power profile using the NVIDIA *nvpmodel* tool that is supplied with the Jetson AGX software stack. This profile fixes the frequencies of the other unused board components (e.g., CV and deep-learning accelerators), ensures that all CPU cores and GPU Texture Processing Clusters (TPC) are *on* all the time, and disables the power gating mechanism that turns some cores *off* when idle. We also disable the system's default DVFS functionality and set all components to the *userspace* governor (which allows us to set the frequencies manually). Moreover, we disable all non-essential OS services running on the boards. Finally, we use the same software stack (e.g., OS, drivers, libraries, etc.) on all boards.

Avoiding thermal throttling. The boards are placed within the same physical rack, hence, they are all in an environment with the same ambient temperature. According to the latest NVIDIA Jetson AGX thermal design guide [36], the maximum operating temperature limits (to operate without performance reduction) for the CPU, GPU, and other components are 86, 88, and 82°C respectively. Above these temperatures software or hardware throttling will reduce runtime frequencies to avoid overheating, thus reducing the board's performance and degrading the reliability of the results. We monitor the on-board temperature sensors to make sure the boards do not enter any thermal throttling zones. Also, we continuously operate the fan at its maximum speed (it usually starts operating at ≈50°C).

6 Results and Analysis

Evaluation metrics. We use six metrics to evaluate the prediction models we develop. The first four represent the percentage of predictions that have relative error lower than a certain threshold (e.g., 5%). The last two metrics are the root mean square error (RMSE) and mean absolute error (MAE) in milliseconds, watts, joules for runtime, power, and energy respectively. We use these multiple metrics to acquire different views of the quality of the models' predictions. Both MAE and RMSE are commonly used in evaluating regression models: MAE gives an unbiased view of the average absolute error, and RMSE places a higher weight on large errors (i.e., penalizes large errors). Thus, both metrics can indicate which models are good candidates for further improvement (e.g., by collecting more training data, or more hyperparameter tuning iterations).

Table 4: Accuracy of kernel-level runtime predictions.

Kernel	5%	10%	15%	20%	RMSE (ms)	MAE (ms)
Conv2d	65.15	82.11	89.97	93.89	12.615	2.785
Batchnorm2d	85.02	95.77	98.11	99.00	0.309	0.121
Matmul	95.25	98.75	99.55	99.76	0.091	0.048
Relu	85.39	95.85	98.04	98.83	0.322	0.135
Maxpool2d	58.52	85.04	93.50	96.60	0.339	0.109
Adaptivepool2d	72.97	92.59	97.13	98.66	0.037	0.015
Add	79.69	92.07	95.85	97.56	0.036	0.007
Cat	49.36	77.00	88.55	93.66	0.336	0.121
Mean	**73.92**	**89.90**	**95.09**	**97.24**	**1.761**	**0.418**

Table 5: Accuracy of kernel-level power predictions.

Kernel	5%	10%	15%	20%	RMSE (W)	MAE (W)
Conv2d	47.08	78.57	91.06	95.61	1.229	0.676
Batchnorm2d	61.68	88.52	92.91	95.40	1.114	0.637
Matmul	72.12	95.68	98.70	99.44	0.566	0.387
Relu	71.10	94.28	97.20	98.18	0.606	0.371
Maxpool2d	57.09	83.94	90.76	94.04	1.209	0.698
Adaptivepool2d	43.95	80.86	93.47	97.80	0.345	0.276
Add	45.39	78.43	91.44	96.68	0.671	0.389
Cat	46.28	74.14	85.80	92.50	1.324	0.747
Mean	**55.59**	**84.30**	**92.67**	**96.21**	**0.883**	**0.523**

The percentage of predictions with errors lower than a certain threshold allows us to: *(i)* more intuitively compare the accuracy of the models between different kernels as well as between runtime and power (as opposed to using the RMSE and MAE values which are averaged metrics), *(ii)* give a better sense of the magnitude of the prediction error as the absolute values for RMSE and MAE become less relevant when predicting over a space with values that span multiple orders of magnitude (as it is the case when exploring the entire frequency configuration space); and *(iii)* more directly map performance of the models to QoS requirements.

6.1 The quality of kernel-level predictions

For each of the eight kernels selected for modeling (§3.2), we train models to predict runtime and power (Tables 4 and 5).

The testing data size differs based on the kernel being evaluated. But, as a rule of thumb, given that we uniformly sample the space (frequency configurations and kernel parameters) and that we dedicate 2 nodes out of 13 for model evaluation, the test data size is usually ≈18% of the total data collected. By dedicating two nodes for evaluation, we make sure that the training data never include any *measurement samples* from these two nodes. The median of each sample is used as the representative ground truth value to calculate the error rates reported in Tables 4 and 5.

We make two key observations from the results presented in Tables 4 (runtime) and 5 (power): first, *the predictions are accurate*: for example, on average (i.e., across all kernels) ≈89.9% of the predictions for the runtime (≈84.3% for power), have errors lower than 10%. Second, even though the convolution kernel runtime model has the largest training dataset (≈1, 400, 000 *measurement samples*), it has lower accuracy than the average kernel. This is due to two factors: *(i)* the complex nature of the convolution kernel - for example, the NVIDIA cuDNN library [13] used by PyTorch to implement the convolution kernel, chooses between seven different implementations depending on the kernel parameters and input shape, and *(ii)* the large parameter space that needs to be sampled.

6.2 The quality of network-level predictions

To make runtime network-level predictions, we use the profiling data gathered during network characterization (§3), we then predict the runtime for each kernel invocation (for the kernels we build models for), and we sum up these values. To evaluate runtime models, we compare the predicted runtimes against the median value of the observed runtimes collected on two boards. To evaluate the power models at the network-level, we need to consider that kernels unevenly contribute to the networks' power consumption. Thus, instead of estimating the network-level power consumption, we estimate the network-level *energy* consumption by summing up the modeled kernels' energy consumption (product of the predicted runtime and power). Then, we compare the estimations against the observed network-level energy consumption.

Fig. 4 highlights the prediction accuracy for runtime (top) and energy (bottom). The figure shows the relative error (compared to *observed* network runtime/energy consumption) for our predictions and two baselines:

- *Our prediction* (left, grey bars in Fig. 4 – labeled *Inference Prediction*). This indicates the relative error of our network-level predictions obtained by aggregating individual, kernel-level *predictions*.
- *Selected kernels observed runtime/energy* (center, white bars) presents the relative (%) error *RE* between the sum of the *selected* kernels' *observed* runtime/energy and the *observed* network inference level runtime/energy. This indicates how close aggregating the kernels' metric of interest (i.e., runtime, energy) measurements is to the network-level inference measurements. The reason to include this baseline is to separate the different sources of error: for this baseline, errors arise because only a selected number of kernels are considered (and no prediction errors impact it). This baseline represents an idealized outcome for our technique based on kernel-level decomposition and kernel selection (as, for each given kernel, its optimal prediction is the observed value).
- *All kernels observed runtime/energy* (right, dark bars) represents the relative error between the sum of *all* kernels' *observed* runtime/energy and the *observed* network runtime/energy. Comparing between the *selected kernels* baseline and this one illustrates the impact of selecting only a subset of the kernels. A big difference would indicate that there are left-out kernels that should have been included in our modeling effort.

The data points in Fig. 4 were collected from two boards at three different *frequency configurations*: low [3], mid, and highest frequencies selected to cover the frequency ranges of the three components (CPU, GPU, and memory controller). There are a number of takeaways: *(i)* overall, with the exception of ShuffleNetV2 (discussed in

[3]Unlike in §3, where for the lowest frequency configuration, the CPU, GPU and memory controller are all set to the lowest possible values, here the CPU is set to a slightly higher frequency than the minimum. The reason is that at the lowest possible CPU frequencies the GPU is underutilized: at these frequencies, the runtime is bottlenecked by the time spent on the CPU preparing for kernel launches, and performing driver related activities while the GPU is idle. A key assumption in our methodology is that the GPU resources are adequately utilized, as such, we measure the time spent by the *GPU kernel activity* and use it as the runtime of the kernels. Hence, any violation of this assumption (such as the behaviour at the lowest CPU frequencies) would require modifying our measurement techniques to account for this overhead as well. This is similar to ShuffleNetV2's behaviour highlighted in §3.2 (though limited to a few frequency configurations unlike for ShuffleNetV2) .

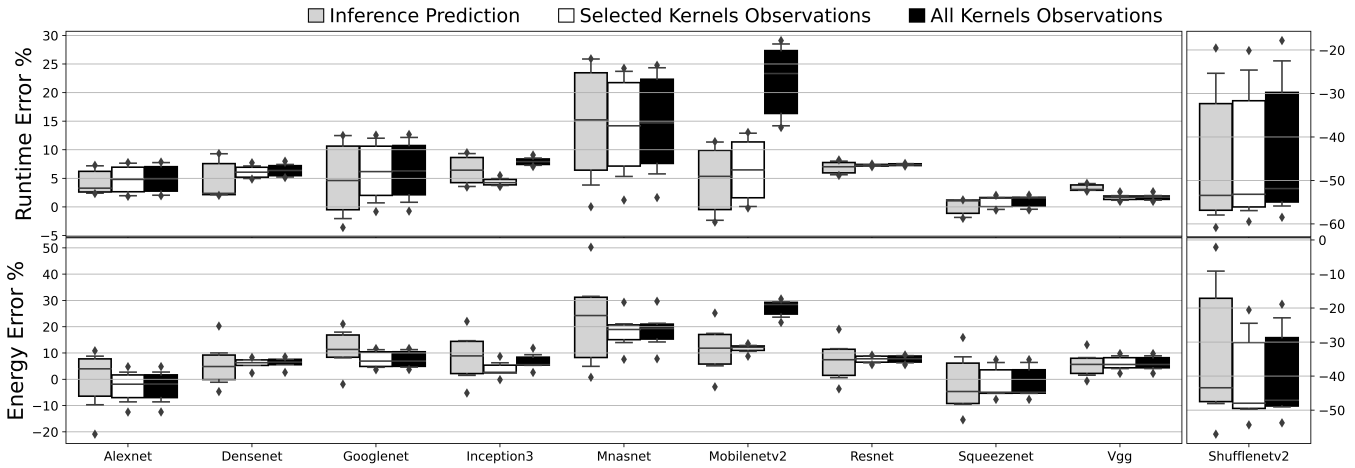


Figure 4: Prediction accuracy for our kernel-level models (grey boxplots) and two baselines: selected kernels' observations (white), and all kernels' observations (black). The x-axis shows the 10 CNNs. The y-axis shows the relative error compared to the observed end-to-end network inference time (top) and energy consumption (bottom). ShuffleNetV2 is separated for better legibility (please see the footnote for ShuffleNetV2's peculiar behaviour). The boxplots are created similar to the previous ones but with outliers represented as black diamonds.

Table 6: Accuracy of our network-level runtime predictions.

Network	5%	10%	15%	20%	RMSE (ms)	MAE (ms)
Alexnet	59.11	92.76	94.86	96.90	1.03	0.82
Densenet	75.33	90.38	93.92	96.48	12.82	8.72
GoogeLeNet	8.43	40.41	52.17	60.19	8.87	6.39
Inception3	24.33	85.49	88.79	91.62	9.50	7.12
Mnasnet	5.73	10.97	16.46	37.08	5.56	4.37
MobileNetV2	8.98	49.19	63.22	67.16	5.01	3.24
Resnet	24.60	60.38	79.60	87.08	3.28	2.45
ShuffleNetV2	3.03	6.56	15.81	21.78	11.48	8.71
Squeezenet	62.73	77.63	80.76	83.87	1.89	1.07
VGG	90.95	93.05	94.90	96.10	4.33	2.82
Mean	**36.32**	**60.68**	**68.05**	**73.83**	**6.38**	**4.57**

Table 7: Accuracy of our network-level energy predictions.

Network	5%	10%	15%	20%	RMSE (J)	MAE (J)
Alexnet	52.43	87.35	97.51	99.54	0.01	0.01
Densenet	46.79	78.59	91.19	97.52	0.13	0.09
GoogeLeNet	19.75	38.41	55.59	68.76	0.04	0.03
Inception3	31.40	61.00	83.33	94.06	0.08	0.05
Mnasnet	6.67	13.35	23.65	35.84	0.03	0.02
MobileNetV2	20.56	41.56	59.41	70.30	0.02	0.02
Resnet	22.59	45.89	68.52	83.19	0.03	0.02
ShuffleNetV2	5.57	12.41	19.00	25.94	0.04	0.03
Squeezenet	45.57	73.73	85.48	90.43	0.01	0.01
VGG	49.86	82.94	93.87	95.83	0.09	0.06
Mean	**30.12**	**53.52**	**67.76**	**76.14**	**0.05**	**0.03**

the footnote), the predictions are accurate; *(ii)* prediction errors do not compose: for all networks, our predictions made by summing up kernel-level predictions are close to the sum of observed kernel behaviour and many errors cancel out during summation; *(iii)* with the exception of MobileNetV2, modeling just the selected kernels is a good approximation for the aggregate behaviour that includes all kernels (as §3.2 suggests).

While Fig. 4 focuses on comparing with the baselines constructed by aggregating *observed* kernel runtime/energy (for three specific frequency configurations only), Tables 6 and 7 summarize the network-level prediction quality for *3,150 frequency configurations*. The key takeaway is that predictions are relatively accurate across the *whole range* of frequency configurations; a range implying more than one order of magnitude variation in runtime or energy. For runtime 60.7% of the predictions have relative error better than 10%; (and 73.8% of the predictions have error lower than 20%). For energy 55.3% of the predictions have relative error better than 10%; (and 77.3% of the predictions error lower than 20%). Finally, the fact that RSME is relatively low suggests that the worst prediction errors are low as well.

6.3 The 'cost' of generality.

Modeling at the kernel level enables generality, that is, making predictions for any CNN whose runtime/energy is dominated by the same set of kernels. To quantify the impact of this methodological decision, we compare it against a specialized approach that models the runtime/energy at the network-level for each individual CNN.

We follow the same training procedure described in §4.3. For each network, on each board (13 boards in total out of which two are reserved for testing), we collect runtime and power *measurement samples* for 3K *frequency configurations*, and use the median of these samples as the target labels during training and testing.

Fig. 5 shows boxplots presenting the distribution of the relative error for both modeling approaches (i.e., generic - at the kernel level, and specialized - at the network level). On average, across all CNNs and all frequency configurations, the network-level modeling approach has lower (better) average relative errors of 3.64% for runtime (4.94% for energy). In contrast, the kernel-level models, have average errors of 15.4% and 14.71% respectively. In absolute terms, adopting a kernel-level modeling approach leads to a 'cost' of increasing the average prediction error by 11.76% for runtime (and 10.22% for energy) but enables a generic prediction engine that can be used to deploy new CNNs without the need to re-train for each network. Whether this cost is acceptable or not would depend

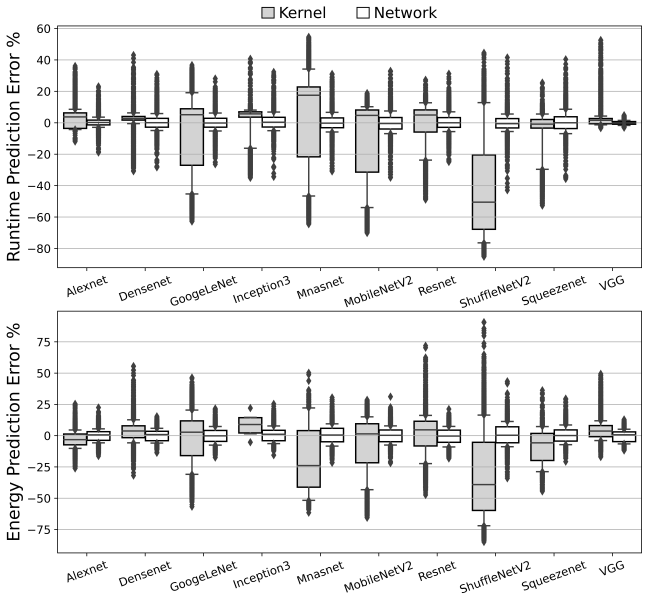


Figure 5: Evaluating the 'cost' of generality. Prediction accuracy comparison between two different network-level modeling approaches: our generic approach aggregating kernel-level models (grey boxplots on the left), and a conventional approach creating a specialized independent model for each network (white boxplots on the right). The y-axis show the relative prediction error percentage *RE* for the runtime (top), and energy consumption (bottom). The boxplots are constructed similar to previous ones (showing quartiles and deciles, with the outliers represented as black diamonds).

on the specific usage scenario (e.g., in situations that explore a large numbers of CNN designs to meet runtime/energy constraints[6, 52] while maximizing accuracy this is likely the only feasible solution as it would be too costly to train models for each independent CNN design considered).

6.4 The impact of considering variability

We conduct an experiment to highlight the impact of taking variability into account. We compare the quality of predictions made by a model trained on data collected from a single board (i.e., the typically used approach: train on node A and deploy on node B) against a model trained on data gathered from multiple boards and tested on an an unseen node (i.e., our proposed approach). We note that most related work (§8) do not take variability into account during both the model development and evaluation phases, where an even less realistic approach is usually used (i.e., train on node A and evaluate on the same node A). This is independent of the underlying modeling choice (e.g., linear regression, decision trees, or neural networks).

On each board, we collect end-to-end inference time and average power (100 observation *measurement samples*) for each of the 10 CNNs for ≈3K *frequency configurations*. We train a single model per network as described in §4.3. For our methodology, we: *(i)* use data collected on 13 boards for training and *(ii)* compare with training on data collected on one node only (and we rotate the training node to obtain 13 predictors). In both cases, we test the quality of the predictions on one remaining dedicated testing board (i.e., a 14^{th} board, randomly chosen). Additionally, we change the testing board once more randomly, and re-run all experiments for a total of 26 comparisons.

Table 8 shows the relative improvement (minimum and maximum values over the 26 comparisons) obtained by training on multiple nodes compared to training on a single node for runtime (left) and power (right). For runtime, on average, the *multi-node model* improves the percentage of predictions with less than 5% error by 6.7% and the RMSE and MAE by 9.5% and 7.2%, respectively. We note that the rate of predictions with larger error percentages (e.g., 20%) does not change much as this is higher than the observed inter-node variability that our model incorporates.

For power the improvements are larger (as expected since inter-node variability is larger as well §3.1): 31.9%, 13.6%, and 14.1% for rate of predictions with better than 10% accuracy, RMSE and MAE respectively. As expected the improvement is higher for the networks where we observe higher inter-node variability (e.g., ShuffleNetV2). We note the particularly large values for the maximum improvement in the rate of predictions with high accuracy (less than 5% error): this is evidence of the fact that a "train on A deploy on B" methodology will generate models that will be particularly inefficient on this success metric on some deployments.

6.5 Training and inference costs.

Training cost. Across the 16 models we developed (two for each of the eight kernels), on a server machine with 2x Intel Xeon E5-2670 v2 (Ivy Bridge) processors, each with 10 cores @ 2.5GHz, the training time min/median/max values are 3.9/4.9/71.2 hours for the runtime models, and 0.8/1.5/5.9 hours for the power models. These relatively low training costs indicate that we have satisfied our last success metric (i.e., training cost is not onerous).

Inference cost. Fig. 6 shows the *relative overhead* - that is, the ratio between the time consumed to make a single prediction (on the ARM CPU of the Jetson AGX), and the predicted runtime (on the on-board GPU). We make predictions for 1,000 randomly generated configurations (i.e., frequencies and parameters) on a per-kernel basis. Then, we construct the boxplots in Fig. 6 to summarize the *relative overhead* values. There are three key observations from the figure: *(i)* the overhead for the power models is much lower than for the runtime models (due to the lower number of boosting trees in the power models); *(ii)* some kernels (e.g., Add and AdaptivePool2D) have a high *relative overhead* due to their low intrinsic computational costs compared to running a complex tree-based model to make predictions; and *(iii)* the *relative overhead* values for the majority of the kernels vary over a wide range. This is because while the cost to make a prediction is almost constant, the predicted runtime is dependent on the frequency configuration.

While Fig. 6 indicates a relatively sizeable inference cost, we note that it can be significantly accelerated. Compiling the models to optimized low-level C libraries [51] is reported to give an over 6× acceleration. One technique we experimented with is grouping prediction in batches as this amortizes some of the prediction overheads. Fig. 7 shows the normalized runtime (batch prediction time divided by the batch size) for the four models with the highest *relative overhead*. The results indicate more than one order of magnitude reduction in the overhead at batch size of 20, and up to

Table 8: Relative improvement (%) in the quality of predictions obtained by taking inter-node variability into account. The tables presents the minimum and the maximum values for the relative improvement observed for runtime (left) and power (right).

Network	Improvement (Min (%) → Max (%)) – RUNTIME			
	5%	10%	RMSE (ms)	MAE (ms)
Alexnet	0.6 → 8.9	-0.0 → 1.9	2.2 → 61.0	-1.1 → 35.6
Densenet	-4.6 → 8.3	-0.8 → 3.4	-0.8 → 9.5	-4.6 → 10.4
GoogeLeNet	-4.9 → 49.8	-0.6 → 18.0	-4.2 → 86.4	-6.7 → 65.5
Inception3	-1.7 → 12.9	-0.1 → 3.5	7.6 → 14.7	0.4 → 11.6
Mnasnet	-5.6 → 31.3	-1.0 → 7.9	-2.2 → 84.5	-10.8 → 62.1
MobileNetV2	-11.5 → 39.8	-1.7 → 9.5	-2.9 → 82.9	-10.1 → 58.9
Resnet	-4.8 → 69.7	-1.1 → 71.2	-0.3 → 82.3	-4.3 → 68.0
ShuffleNetV2	-12.6 → 186.3	-1.7 → 13.1	-3.8 → 91.4	-19.0 → 75.7
Squeezenet	-0.5 → 16.3	0.7 → 10.6	-0.1 → 78.3	-0.3 → 51.1
Mean	**-5.1 → 47.0**	**-0.7 → 15.5**	**-0.5 → 65.7**	**-6.3 → 48.8**

Network	Improvement (Min (%) → Max (%)) – POWER			
	5%	10%	RMSE (W)	MAE (W)
Alexnet	-27.3 → 1,268.0	-3.2 → 80.9	-82.7 → 52.6	-98.3 → 55.2
Densenet	-19.6 → 442.4	-4.4 → 78.4	-51.1 → 47.2	-51.9 → 50.9
GoogeLeNet	-24.2 → 1,006.1	-2.9 → 162.2	-104.4 → 57.2	-101.8 → 62.2
Inception3	-25.5 → 465.6	-5.6 → 99.4	-56.7 → 45.2	-50.7 → 52.9
Mnasnet	-22.6 → 1,894.3	-6.0 → 145.2	-67.5 → 55.6	-69.6 → 60.9
MobileNetV2	-14.8 → 1,207.4	-3.5 → 140.4	-53.8 → 59.1	-41.2 → 65.0
Resnet	-28.9 → 468.0	-8.1 → 107.0	-72.5 → 43.8	-74.6 → 50.2
ShuffleNetV2	-39.6 → 113,950.0	-9.0 → 755.6	-70.3 → 66.2	-77.7 → 71.0
Squeezenet	-27.1 → 582.4	-5.0 → 95.0	-48.6 → 46.1	-42.9 → 56.0
Mean	**-25.5 → 1,3476.0**	**-5.3 → 184.9**	**-67.5 → 52.5**	**-67.6 → 58.3**

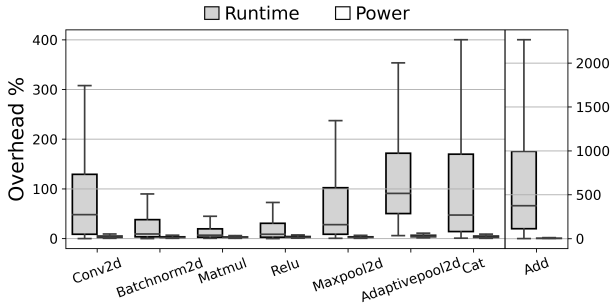


Figure 6: Relative inference overhead for kernel-level models. The y-axis represents the relative overhead (i.e., the ratio between the time it takes to obtain a prediction using the model and the predicted kernel runtime.

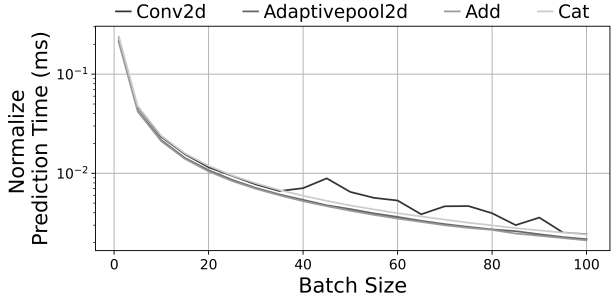


Figure 7: Batch size impact on the models' prediction performance. The x-axis shows the batch size and the y-axis (log scale) shows the prediction time (ms) normalized to the batch size.

two orders of magnitude at 100 batch size.

The key takeaway is that several techniques are available to reduce the runtime cost of the models such that they can be used in an online adaptation scenario which is the second success criteria.

7 Discussion

We discuss several interrelated topics:

Generality of our modeling approach. While the models we developed and evaluated (in §6.1 and §6.2) are specific to the AGX platform, our modeling methodology (§4) is generic and applicable to other platforms as it does not use any platform-specific information or insight. We model the runtime (and power) for the most widely used kernels (in deep-learning workloads, using only the input shape and kernel parameters; hence, one can simply apply our methodology on a different target platform by collecting runtime (and power) measurements at different frequency configurations on that target platform, using the same kernel-level information, and then re-train the models.

Variability characterization in edge computing environments. We stress that previous work for characterizing inter-node variability and, for identifying its sources (e.g., manufacturing process, hardware aging, software-level interference, etc.) exists in HPC, Cloud, and mobile environments (§8). Yet, variability remains widely ignored in Edge computing environments. This paper sheds light on this problem and presents, to the best of our knowledge, a first attempt to systematically study variability on Edge computing devices, and how to address it using a simple technique in the modeling context. We believe that our approach for variability characterization is generally applicable and can be extended to other computing environments, but that remains out of the scope of this paper.

Profiling techniques are important yet often overlooked. To the best of our knowledge, there is no standard for profiling runtime and power consumption on recent hardware such as the Jetson AGX. This leads to an often overlooked problem: *a fragmentation in the profiling techniques that target the same platform.*

For example, to estimate the runtime of heterogeneous applications that run on CPUs and GPUs, there are several decisions to be made such as: *(i)* whether to include memory copying to/from the GPU, and the CUDA synchronization calls when measuring the GPU kernels' runtime along with their CPU launching overhead, *(ii)* whether it is practical to leverage the available NVIDIA tools solely (e.g., NVPROF) to measure the CUDA kernels' runtime on the GPU, and *(iii)* how many timing and warm up iterations to run, whether to report the average or the median of the collected observations, and which one to use as a representative value for the measured runtimes, etc.

At the same time there are factors beyond the developer control such as the precision, accuracy, and overhead of the measurement tools (e.g., CUDA events, CPU clock, timing libraries, internal/external power sensors). These factors affect the reproducibility of prior work, and from our experience, they have a large impact on the quality and reliability of a prediction engine. As we describe in §5.2, we went to great lengths to ensure the accuracy of our timing measurements for example, by using CUDA events.

The manifestations of inter-node variability are non-uniform. §3.1 highlights the magnitude of inter-node variability and compares it to intra-node variability. An important lesson is that inter-node variability does not follow a predictable trend across all networks. We believe this is a consequence of two factors: *(i)* different CNNs stress different system resources (e.g., CPU, GPU, and memory), and *(ii)* inter-node variability is not uniform across those nodes' resources. In other words, inter-node variability is a 2D problem (i.e., parametrized by the nodes' resources and the application workload), which makes it more challenging to deal with it.

Trade-offs: models' accuracy versus their inference and training costs. As detailed in §4.3, there are several hyperparameters that control the complexity of the models (e.g., number of estimators, tree depth, etc.). More complex models may lead to higher accuracy but at the cost of larger memory footprints and slower inference rates. Hence, there is a trade-off between the runtime overhead of the models and the target accuracy. Similarly, during the hyperparameter tuning phase, one specifies the number of tuning iterations used to search for the best hyperparameters. A larger number of iterations increases the probability that a better set of hyperparameters is found; however, it increases the training time for a fixed resource limit (same allocated training cores/machines). One consequence of these observations is that, while we have met our success criteria (§1), one can tune the models' accuracy and overhead to meet other modified criteria suited for a different deployment scenario.

Lowest CPU frequencies lead to degraded energy efficiency. While the Jetson AGX features 29 different CPU frequencies, we find that the lowest CPU frequencies are usually particularly inefficient in terms of overall energy efficiency for heterogeneous applications. The reason is that in these applications the CPU drives the workload on the other accelerators (e.g., GPU, DL, PVA), hence, it is essential that the CPU does not become the bottleneck where the accelerators are idle / underutilized waiting for work to be scheduled on their queues. This issue was apparent in §6.5 and afterwards, we ignore the first few (lowest) CPU frequencies.

8 Related work

This section discusses the most relevant studies that relate to this work. We note that none of the studies we are aware of takes into account inter-node variability: almost all studies train models and test them on the same board (rarely on the same *set* of boards, yet without differentiating between a training and test set).

Modeling and characterization. Bouzidi et al. [9] propose a network-level modeling approach for predicting CNN inference runtime. Their objective is to develop a runtime prediction tool that allows developers to choose the optimal CNN given a specific edge platform architecture. The study focuses solely on runtime. The key novelty relates to feature engineering: the study demonstrates that a small number of features (e.g., generated FLOPS rate, number of convolution layers, neurons and weights, input sizes, etc.) is sufficient to build a model that can be used on unseen CNNs (once the deployment space has been characterized). The key differences from our approach are: *(i)* a different motivating scenario; and *(ii)* different modeling granularity (network-level as opposed to kernel-level); and *(iii)* the authors do not attempt to predict the metrics of interest for different *frequency configurations*.

Lu et al. [31, 32] study CNNs' resource requirements (e.g., runtime, memory, and power) on mobile devices. They characterize the CNNs at layer-level granularity and develop a model to predict the CNNs' core computational resource requirements. Then, they combine the characterization and model to build a prediction tool *Augur*, that takes as input a CNN description, and predicts its runtime, memory and power consumption. There are two key differences between their approach and ours. Firstly for the deployment, limited layer selection, and CNNs the authors study, matrix multiply is the dominant core computation. Thus, their approach relies on modeling matrix multiply computation to estimate the layer-level performance. While this approach simplifies the modeling phase, it has two shortcomings: *(i)* in cases where the network computations are not bottlenecked by matrix multiplication, their approach leads to inaccurate approximation (e.g., 35% and 45% on the CPU and GPU for GoogLeNet's runtime); *(ii)*, modeling only matrix multiplication is insufficient for other networks (e.g., SqueezeNet), that frequently call other kernels. Secondly, in the most recent version of Augur [32], the system's frequencies are fixed at the maximum values for better performance predictability, and hence, the modeled space is reduced by a factor of 3,600X (i.e., number of *frequency configuration*) on the Jetson AGX.

Davis et al. [15] are the closest project to our work from the perspective of considering variability: they highlight inter-node variability impact on building power models based on hardware counters in large-scale homogeneous clusters. They note the variability among the multiple nodes of a cluster and propose a similar approach to ours (i.e., sampling training data from multiple cluster nodes) to account for variability while building power models. Apart from the different models being built, there are a number of other differences compared to our work: *(i)* three of the four platforms studied had the DVFS governor running on default system policy (i.e., DVFS configuration is dynamically changed depending on the underlying resources utilization level), which means that the dynamic frequency selection contributed to the observed variability, *(ii)* the experimental setup relied on the CPU solely (i.e., leading to a much smaller DVFS configuration space compared to the Jetson AGX), *(iii)* the power models are developed based on low-level features (i.e., performance counters) collected during applications runtime, and *(iv)* less emphasis is put towards controlling potential sources of variability (e.g., DVFS settings, dynamic power management, concurrent applications, ambient temperature, etc.).

End-to-end optimization solutions. Wan et al. [52] present Alert, a runtime scheduler that relies on a feedback-based controller and probabilistic model to select a DNN and the frequency configurations of the system's resources. The goal is to achieve specific latency, accuracy, and energy objectives. Alert presents a cross-stack approach to adaptation, in which it combines DNN selection (determines the accuracy) with system tuning (frequency selection) to meet the QoS objectives. Alert addresses runtime variability on the same device (intra-node) that occurs due to different inputs, concurrent activity, and power management.

Baruah et al. [5] propose Airavat, a framework that manages the frequency configuration on heterogeneous embedded devices (they experiment on Jetson TX1). The main objective is to minimize energy consumption (by optimizing for energy-delay product). Prediction models are trained offline employing a Random Forest (RF)

model. Developing these models requires three key steps: *(i)* collecting a large set of representative applications, *(ii)* profiling these applications, at different frequency configurations, using low-level performance counters to measure and collect metrics of interest (e.g., L1 miss rate, Memory Access, Cycles Per Instruction), and *(iii)* identifying the settings that result in the lowest EDP. In addition to the offline trained models, they propose an online tuning layer that combines runtime performance counters and utilization metrics to fine-tune the frequency scaling.

9 Summary and Future work

We explored the feasibility of building a robust engine for predicting the runtime and power/energy consumption of CNN inference workloads executed on the Jetson AGX. The modeling approach we propose is unique as a result of two methodological decisions both targeted to address the challenges of realistic deployment scenarios:

- First, we take into account the variability among *nominally identical* edge platforms. This has two advantages: *(i)* it improves the average quality of our predictions; and *(*ii) when the model is deployed across multiple nodes, it reduces the variability in the quality of its predictions. (§3.1, §6.4)
- Second, while for DNNs, modeling at the network and layer-level granularity have been used before, we show the feasibility of a more fine-grained and flexible approach, i.e., modeling at the kernel-level granularity. This allows us to develop a *generic* prediction engine that can be used to make predictions for any DNN (or layer) that is dominated by a small set of kernels. (§4.2, §6.3)

The resulting models for runtime and power/energy have the properties we set out to obtain (§6). They are *(i) effective* - with average relative predictions error of 15.4% for runtime (and 14.86% for energy) and low RMSE; *(ii) efficient* - indicating the ability to make predictions at a fraction of the runtime cost (after optimizing the Python code generated by XGBoost); *(iii) generic* - making it possible to deploy updated/new inference models (or using different type of input data - e.g., changing image resolution) without model retraining, and *(iv) practical* - with low training cost.

Future work. We plan to explore several research directions moving forward:

- First, *extend the scope of the variability characterization*. We plan to include: *(i)* more edge device designs, and *(ii)* low-level benchmarks (e.g., Rodinia [11]). The preliminary results from our early experiments on the Jetson Nano edge platform and using benchmarks from Rodinia are inline with our findings in this paper on the AGX platform and using PyTorch deep-learning networks.
- Second, *investigate the likely sources of the observed variability*. In this context, we believe there are several sources worth exploring (e.g., hardware, frequency choice, workload characteristics, OS scheduling decisions, etc.).
- Third - *enhance runtime and power models' accuracy and performance*. While the presented models met the success criteria we defined earlier (§1), we believe there is room for improvement in accuracy and overhead especially for some networks (e.g., MnasNet). This can be achieved by: careful investigation of the sources of models' errors, sampling more data points, and exploring alternative modeling techniques (e.g., deep neural networks).
- Fourth - *evaluating the impact of variability on an end-to-end controller*. We plan to evaluate the impact of accounting for variability on the performance of an end-to-end model-based frequency controller. Traditionally, these controllers are based on prediction models (or lookup tables) that are trained and tested on data collected from the same node [5, 7, 8, 38, 53]. However, as we show in §8, accounting for variability significantly improves the models' accuracy. Subsequently, this will impact the behavior of a frequency controller that relies on predictions made by these models to make runtime decisions.

Acknowledgments

This project was sponsored in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by a gift fund from Huawei Toronto Heterogeneous Compiler Lab in Canada. We also thank the reviewers of SEC '21 for their insightful feedback. And finally, we thank our shepherd, professor Z. Morley Mao, for her guidance during the preparation of the camera-ready version of the paper.

References

[1] Martín Abadi, Paul Barham, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Ossama Abdel-Hamid, Abdel-rahman Mohamed, et al. 2014. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing* 22, 10 (2014), 1533–1545.

[3] Hazem A. Abdelhafez, Hassan Halawa, et al. 2021. Snowflakes at the Edge: A Study of Variability among NVIDIA Jetson AGX Xavier Boards. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3434770.3459729

[4] Robert Adolf, Saketh Rama, et al. 2016. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[5] Trinayan Baruah, Yifan Sun, et al. 2018. Airavat: Improving energy efficiency of heterogeneous applications. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE*, Jan Madsen and Ayse K. Coskun (Eds.). IEEE, Dresden, Germany, 731–736. https://doi.org/10.23919/DATE.2018.8342104

[6] Soroush Bateni and Cong Liu. 2020. NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 371–385. https://www.usenix.org/conference/atc20/presentation/bateni

[7] Soroush Bateni, Husheng Zhou, et al. 2018. PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 107–118. https://doi.org/10.1109/RTSS.2018.00020

[8] Josep Ll. Berral, Íñigo Goiri, et al. 2010. Towards Energy-Aware Scheduling in Data Centers Using Machine Learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking* (Passau, Germany) *(e-Energy '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1791314.1791349

[9] Halima Bouzidi, Hamza Ouarnoughi, et al. 2021. Performance Prediction for Convolutional Neural Networks on Edge GPUs. In *Proceedings of the 18th ACM International Conference on Computing Frontiers* (Virtual Event, Italy) *(CF '21)*. Association for Computing Machinery, New York, NY, USA, 54–62. https://doi.org/10.1145/3457388.3458666

[10] Zhaowei Cai, Quanfu Fan, et al. 2016. A Unified Multi-scale Deep Convolutional Neural Network for Fast Object Detection. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 354–370.

[11] Shuai Che, Michael Boyer, et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[12] Tianqi Chen and Carlos Guestrin. 2016. XGBoost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug 2016). https://doi.org/10.1145/2939672.2939785

[13] The cuDNN Team. 2021. *NVIDIA CUDA Deep Neural Network library (cuDNN)*. NVIDIA. Retrieved January, 2021 from https://developer.nvidia.com/cudnn

[14] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 175–180. https://doi.org/10.1109/UCC-Companion.2018.00053

[15] John Davis, Suzanne Rivoire, et al. 2011. Accounting for Variability in Large-Scale Cluster Power Models. In *2nd Workshop on Exascale Evaluation and Research Techniques, Held in Conjunction with ASPLOS 2011* (2nd workshop on exascale evaluation and research techniques, held in conjunction with asplos 2011 ed.). Association for Computing Machinery, Inc. https://www.microsoft.com/en-us/research/publication/accounting-for-variability-in-large-scale-cluster-power-models/

[16] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 1437–1446. http://proceedings.mlr.press/v80/falkner18a.html

[17] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232. http://www.jstor.org/stable/2699986

[18] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational statistics & data analysis* 38, 4 (2002), 367–378.

[19] Benjamin Gaudette, Carole-Jean Wu, and Sarma Vrudhula. 2016. Improving smartphone user experience by balancing performance and energy with probabilistic QoS guarantee. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 52–63.

[20] Puneet Gupta, Yuvraj Agarwal, et al. 2012. Underdesigned and opportunistic computing in presence of hardware variability. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems* 32, 1 (2012), 8–23.

[21] Tianshu Hao, Yunyou Huang, et al. 2018. Edge AIBench: towards comprehensive end-to-end edge computing benchmarking. In *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 23–30.

[22] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering* 21, 9 (2009), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

[23] John L Hodges. 1958. The significance probability of the Smirnov two-sample test. *Arkiv för Matematik* 3, 5 (1958), 469–486.

[24] Forrest N. Iandola, Song Han, et al. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. arXiv:1602.07360 [cs.CV]

[25] Guolin Ke, Qi Meng, et al. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.

[26] Brian Kocoloski and John Lange. 2018. Varbench: An Experimental Framework to Measure and Characterize Performance Variability. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/3225058.3225125

[27] Steve Lawrence, C Lee Giles, et al. 1997. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks* 8, 1 (1997), 98–113.

[28] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.

[29] Lisha Li, Kevin Jamieson, et al. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 6765–6816.

[30] Xiaqing Li, Guangyan Zhang, et al. 2016. Performance analysis of GPU-based convolutional neural networks. In *2016 45th International conference on parallel processing (ICPP)*. IEEE, 67–76.

[31] Zongqing Lu, Swati Rallapalli, et al. 2017. Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices. In *Proceedings of the 25th ACM International Conference on Multimedia* (Mountain View, California, USA) *(MM '17)*. Association for Computing Machinery, New York, NY, USA, 1663–1671. https://doi.org/10.1145/3123266.3123389

[32] Zongqing Lu, Swati Rallapalli, et al. 2021. Augur: Modeling the Resource Requirements of ConvNets on Mobile Devices. *IEEE Transactions on Mobile Computing* 20, 2 (2021), 352–365. https://doi.org/10.1109/TMC.2019.2946538

[33] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[34] John C. McCullough, Yuvraj Agarwal, et al. 2011. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, Vol. 20.

[35] Alexey Natekin and Alois Knoll. 2013. Gradient boosting machines, a tutorial. *Frontiers in Neurorobotics* 7 (2013), 21. https://doi.org/10.3389/fnbot.2013.00021

[36] NVIDIA. 2020. *Jetson AGX Xavier Series: Thermal Design Guide*. https://tinyurl.com/r7zeehya

[37] NVIDIA. 2021. *NVIDIA Visual Profiling Toolkit*. NVIDIA. Retrieved Accessed: January, 2021 from https://developer.nvidia.com/nvidia-visual-profiler

[38] Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2017. ML-Gov: A Machine Learning Enhanced Integrated CPU-GPU DVFS Governor for Mobile Gaming. In *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia* (Seoul, Republic of Korea) *(ESTIMedia '17)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/3139315.3139317

[39] Adam Paszke, Abhishek Chaurasia, et al. 2016. Enet: A deep neural network architecture for real-time semantic segmentation. *arXiv preprint arXiv:1606.02147* (2016).

[40] Adam Paszke, Sam Gross, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

[41] Liudmila Prokhorenkova, Gleb Gusev, et al. 2018. CatBoost: Unbiased Boosting with Categorical Features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6639–6649.

[42] Vijay Janapa Reddi, Christine Cheng, et al. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[43] F. W. Scholz and M. A. Stephens. 1987. K-Sample Anderson–Darling Tests. *J. Amer. Statist. Assoc.* 82, 399 (1987), 918–924. https://doi.org/10.1080/01621459.1987.10478517

[44] Wenzhe Shi, Jose Caballero, et al. 2016. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1874–1883.

[45] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[46] C. Szegedy, Wei Liu, et al. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[47] The Jetson Platform Team. 2021. *NVIDIA Jetson AGX Xavier Specifications*. NVIDIA. Retrieved January, 2021 from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

[48] The PyTorch Team. 2021. *CUDA Timing Events*. https://pytorch.org/docs/stable/cuda.html#torch.cuda.Event

[49] TI. 2021. *INA3221 Power Monitors*. Texas Instruments. https://www.ti.com/product/INA3221

[50] Peter Torelli and Mohit Bangale. [n.d.]. Measuring inference performance of machine-learning frameworks on edge-class devices with the mlmark benchmark. *Techincal Report. Available online: https://www. eembc. org/techlit/articles/MLMARK-WHITEPAPERFINAL-1. pdf (accessed on October 2021)* ([n. d.]).

[51] TreeLite. 2021. *TreeLite Documentation*. https://treelite.readthedocs.io/en/latest/

[52] Chengcheng Wan, Muhammad Santriaji, et al. 2020. ALERT: Accurate Learning for Energy and Timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 353–369. https://www.usenix.org/conference/atc20/presentation/wan

[53] Siqi Wang, Gayathri Ananthanarayanan, et al. 2020. High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2254–2267. https://doi.org/10.1109/TCAD.2019.2944584

[54] Hannes Weisbach, Balazs Gerofi, et al. 2018. Hardware Performance Variation: A Comparative Study Using Lightweight Kernels. In *High Performance Computing*. Springer International Publishing, Cham, 246–265.

[55] Carole-Jean Wu, David Brooks, et al. 2019. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

[56] XGBoost. 2021. *XGBoost Documentation*. https://xgboost.readthedocs.io/en/latest/parameter.html

[57] Wei Zhang, Wei Wei, et al. 2019. Ai matrix: A deep learning benchmark for alibaba data centers. *arXiv preprint arXiv:1909.10562* (2019).