A Case for Generalizable DNN Cost Models for Mobile Devices

Vinod Ganesan*, Surya Selvam*, Sanchari Sen†, Pratyush Kumar* and Anand Raghunathan†
* Department of Computer Science and Engineering, IIT Madras, India
† School of Electrical and Computer Engineering, Purdue University
{vinodg,cs16b029,pratyush}@cse.iitm.ac.in, {sen9,raghunathan}@purdue.edu

***Abstract*—Accurate workload characterization of Deep Neural Networks (DNNs) is challenged by both network and hardware diversity. Networks are being designed with newer motifs such as depthwise separable convolutions, bottleneck layers, *etc.*, which have widely varying performance characteristics. Further, the adoption of Neural Architecture Search (NAS) is creating a Cambrian explosion of networks, greatly expanding the space of networks that must be modeled. On the hardware front, myriad accelerators are being built for DNNs, while compiler improvements are enabling more efficient execution of DNNs on a wide range of CPUs and GPUs. Clearly, characterizing each DNN on each hardware system is infeasible. We thus need cost models to estimate performance that generalize across both devices and networks. In this work, we address this challenge by building a cost model of DNNs on mobile devices. The modeling and evaluation are based on latency measurements of 118 networks on 105 mobile System-on-Chips (SoCs). As a key contribution, we propose that a hardware platform can be represented by its measured latencies on a judiciously chosen, small set of networks, which we call the signature set. We also design a machine learning model that takes as inputs (i) the target hardware representation (measured latencies of the signature set on the hardware) and (ii) a representation of the structure of the DNN to be evaluated, and predicts the latency of the DNN on the target hardware. We propose and evaluate different algorithms to select the signature set. Our results show that by carefully choosing the signature set, the network representation, and the machine learning algorithm, we can train accurate cost models that generalize well. We demonstrate the value of such a cost model in a collaborative workload characterization setup, wherein every mobile device contributes a small set of latency measurements to a centralized repository. With even a small number of measurements per new device, we show that the proposed cost model matches the accuracy of device-specific models trained on an order-of-magnitude larger number of measurements. The entire codebase is released at https://github.com/iitm-sysdl/Generalizable-DNN-cost-models.**

***Index Terms*—runtime, deep learning, mobile devices, machine learning**

I. Introduction

Deep Neural Networks (DNNs) have achieved remarkable success in a wide range of domains. As they get deployed on resource-constrained platforms, the focus in designing DNNs is shifting from designing accurate networks to designing *accurate and efficient* networks. On the one hand, this is challenged by the need to characterize DNNs on a wide variety of hardware devices - different generations of CPUs, GPUs and the rapidly increasing set of accelerators. On the other hand, there is a growing diversity of neural networks due to newer design motifs such as skip connections, depthwise separable convolutions, bottleneck layers, *etc*. Further, the adoption of Neural Architecture Search (NAS) [1], [2] is creating a Cambrian explosion in networks with characteristics that often differ from human-designed networks. This product space of DNNs and hardware platforms is increasing at a scale where explicitly measuring the run-time of each network on each hardware is too expensive. It is thus critical to create *cost models* that can estimate latency for a given network and hardware platform. Unfortunately, existing efforts on such cost models are restricted to either a limited space of networks (*e.g.*, in the context of fine-tuning a base network structure), or a limited set of hardware platforms. Further, most existing studies train separate cost models for each hardware platform [2]–[5], requiring a large number of measurements across hardware. Thus, such approaches may not be feasible in the context of large-scale deployment of DNNs in practice. For example, the developer of a DNN-enabled mobile app may want to estimate latency on the wide range of devices on which the app may be installed without the ability to explicitly characterize on each of them.

In this paper, we systematically study DNN cost models on a large collected data-set consisting of run-times of 118 networks on 105 mobile devices. The 118 networks consist of popular DNNs for computer vision applications as well as randomly generated DNNs with varying number of layers, operators, filter-sizes, and channels. The 105 mobile devices, which are largely obtained by crowd-sourcing, represent CPUs used in more than 72% of today's mobile devices [6]. On each device, we measure the latency of each network executed 30 times using a custom Android app.

With exploratory data analysis of the variation in measured latencies across networks and devices, we establish the need for good cost models. We then focus on an essential aspect of a generalizable cost model: the input representation for both the network and the hardware platform. We represent each network by encoding layer-wise its operators (Conv, ReLU, *etc*.) and parameters (kernel size, stride, padding, *etc*.) as a vector. For the latter, we find that representing a hardware platform by basic parameters like core frequency and main

Surya Selvam is currently a PhD student at Purdue University (selvams@purdue.edu)
Sanchari Sen is currently a research scientist at IBM T.J. Watson Research Center, Yorktown Heights, NY (sanchari.sen@ibm.com)

978-1-7281-7645-1/20/$31.00 ©2020 IEEE
DOI 10.1109/IISWC50251.2020.00025

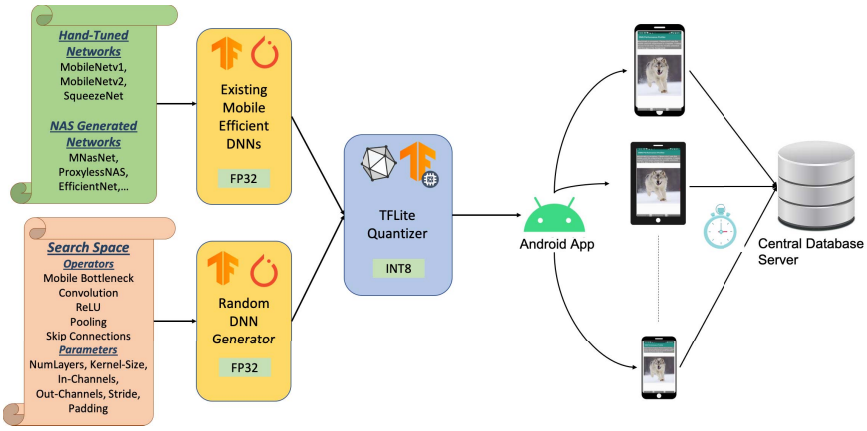

Fig. 1. Characterization framework for DNNs on mobile devices. Our network set includes both randomly generated DNNs as well as hand-designed [7]–[9] and NAS generated [1], [2], [10], [11] DNNs. We quantized each DNN to 8-bits using TFLite's post-training quantizer. We developed an Android app with the TFLite interpreter to execute the DNNs on the devices. The app schedules each DNN on the CPU and measures the inference latency, averaged across 30 runs. The measured values along with some queried hardware parameters are sent over HTTP to a database. We crowd-sourced the application and collected data on 105 mobile devices.

memory size alone leads to poor accuracy. One way to improve accuracy is to use detailed micro-architectural features to represent the hardware. However, such details may not always be available to a software developer. Instead, we propose an alternative approach of representing each hardware platform by the run-times of a small signature set of networks on it.

To maximize the model's accuracy, we evaluate three sets of methods to select the signature set: random sampling (RS), Mutual Information based sampling (MIS), and Spearman Correlation Coefficient based sampling (SCCS). We use XGBoost, a state-of-the-art ML algorithm based on gradient boosting, as the regression method to estimate runtime. In our experiments, we found that even a very small signature set of up to 10 networks leads to cost models with high accuracy.

We further observe that the learnt cost model generalises well across networks and hardware platforms. However, generalisation is weaker when the cost model is tested on an adversarially chosen small set of hardware platforms.

In summary, we present the first systematic study of cost models for estimating run-times of DNNs across a wide-range of mobile devices. Our analysis suggests a simple, yet effective, method for building generalizable cost models through collaboration:

- Maintain a *repository* of the run-times on different devices of a small signature set of up to 10 networks.
- Use these run-times as a representation for each device to train a cost model on a larger set of networks.
- For a new network, use the cost model to predict the run-time on any device with a representation in the repository.
- For a new device, measure the run-time of the signature set and add it to the repository.
- Periodically fine-tune the cost model as more measurements are added in the repository.

With our collected data, we simulate such a collaborative framework. We find that the cost model trained on the repository matches the accuracy of device-specific models trained on an order-of-magnitude larger number of measurements. Building and maintaining such a global framework would be of interest to both network and hardware designers and would significantly benefit the process of designing and deploying efficient DNNs. Additionally, such a cost-model could significantly improve the search-time, and even the performance, of hardware-aware Neural Architecture Search algorithms [2], [12] and domain-specific compilers [3], [13] across a wide-variety of hardware devices.

The rest of the paper is organized as follows. In Section II, we detail the procedure used to collect data across mobile devices, and share results of exploratory data analysis on the collected measurements. In Section III, we describe our core contribution of generalizable cost models. We then present and discuss experimental results in Section IV. We illustrate collaborative workload characterisation in Section V. We discuss related work in Section VI and conclude in Section VII.

II. Dataset Collection and Exploratory Analysis

In this section, we first describe our approach to collecting run-time measurements across devices, which consists of two components: a parameterized DNN generator and an Android app framework. Figure 1 graphically illustrates these components. We then perform exploratory data analysis to reveal distributional and relational properties across the data. Our analysis suggests that basic hardware features of a device are insufficient for predicting latency across DNNs.

A. Parameterized DNN Generator

We perform the run-time measurements on mobile devices across a benchmark suite of neural networks, consisting of 18 popular pre-designed networks and 100 randomly generated networks. The first set of networks includes hand-tuned networks such as MobileNets [7], [8] and SqueezeNet [9], as well as networks generated with Neural Architecture Search

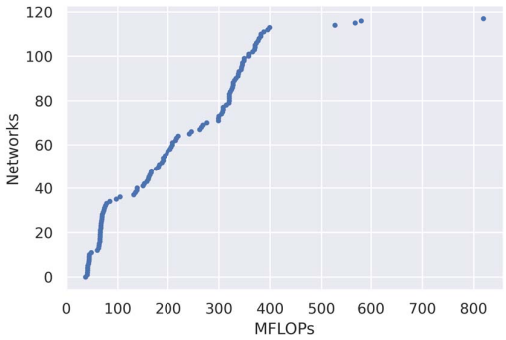

Fig. 2. Distribution of FLOPs (in millions) for the 118 networks. The FLOPs of the networks range from 400 million MACs to 800 million MACs

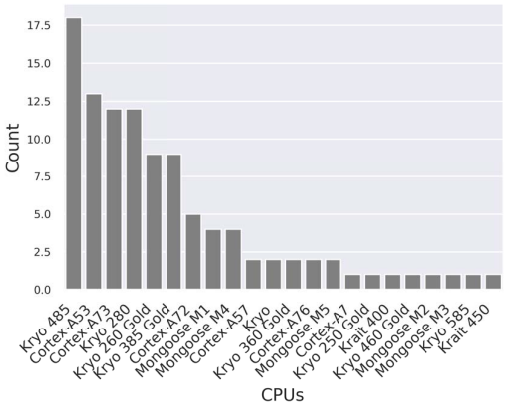

Fig. 3. Histogram of CPUs for the 105 devices. The devices show significant diversity from Cortex-A53, which is almost eight years old, to Kryo-585, a few months old. These diverse CPUs capture various micro-architectural and technology changes across processor generations.

(NAS) [1], [2], [10], [11]. The randomly generated networks in the second set are obtained using an in-house parameterized DNN generator — a PyTorch [14] framework to generate arbitrary but valid DNNs within a user-defined search space. This search space has been adapted from popular hardware-aware NAS frameworks [2], [11], [12] and is composed of different operators and parameters, as illustrated in Figure 1. The operator set covers the spectrum of network design patterns used in mobile DNNs including inverted bottleneck layers, convolutions, activations, pooling and skip-connections. The parameter set captures network features such as number of layers, kernel size, number of input and output channels, stride, padding, groups *etc*. These networks are diverse: We visualize this diversity along the axis of FLOPs required to run inference on these networks in Figure 2.

B. Android App for Latency Measurement

To measure the latency on a large set of devices, we chose to develop an Android app and crowd-source the measurements. Our Android app uses the TFLite [15] runtime APIs to measure the inference latency of DNNs, as it is competitive in terms of performance among mobile DNN frameworks. Since our DNN generator outputs PyTorch networks, we first convert them to TFLite models using ONNX [16]. In order to optimize the application size and encourage more users to participate in the crowd-sourcing, we quantized the networks to `int8` precision using TFLite's post-training quantizer, as shown in Figure 1. Such quantization is routinely performed and represents the typical deployment procedure for mobile devices [17].

Many mobile devices contain accelerators such as GPUs, DSPs and even Neural Processing Units (NPUs), in addition to CPUs. However, a majority of edge inference workloads still run on low-power CPUs, while only a fraction of them are run on GPUs and NPUs [6]. Further, programmability of these available GPUs and NPUs is a major bottleneck. We observed that the GPU and NPU Android API delegates were either limited to a certain class of mobile phones or were prone to unexpected outcomes (very high latency) or crashes. To ensure reliable collection of measurements across a large set of mobile devices, we restrict our focus to mobile CPUs. However, the methodology presented in the subsequent sections would also apply to execution on GPUs and NPUs.

Our Android application sequentially schedules each of the 118 quantized DNNs to one of the big CPU cores in the mobile's big.Little CPUs, similar to the technique followed by the authors of MobileNetV3 [12]. We measure the single-threaded inference latency of all the networks on a single large CPU with a batch size of one and average the values across 30 runs. The values are automatically transmitted to a public server via HTTP upon completion of the application.

Our application was made available publicly (https://hwcostmodels.github.io), allowing us to gather measurements of 118 networks on 105 mobile devices, resulting in a total of 12,390 data points (each of which is a mean of 30 measurements). Figure 3 shows the distribution of CPUs in this set. As can be seen, there is a large diversity of devices across multiple chipsets (38 unique types), and core families (22 unique types). We estimate that these devices represent the CPUs in roughly 72% of mobile device in the field today [6].

C. Exploratory Data Analysis

We next present results of exploratory data analysis on the collected measurements. Our objectives in this analysis are twofold. One, to visualize the variation of latency across networks and hardware and to thereby identify clusters of similar hardware devices and networks. Two, to identify any obvious patterns that relate the hardware and network features to the measured latencies.

Clustering Devices. To begin, we cluster the hardware devices where each device is represented by a 118 dimensional vector corresponding to the latency on all networks. We use the standard k-means clustering algorithm with different values of k (the number of clusters). We find that k = 3 is a good choice dividing the devices into three clusters which we call *fast*, *medium*, and *slow*, since they demonstrate mean latencies of

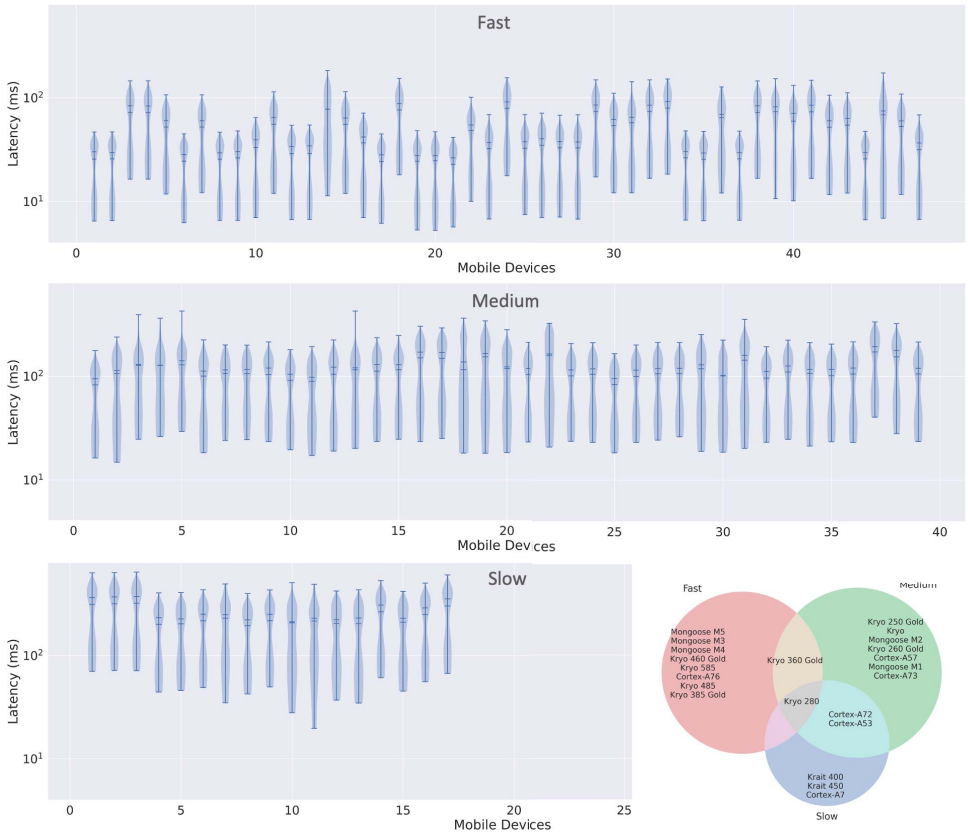

Fig. 4. 105 hardware devices categorized into 3 clusters: *fast*, *medium* and *slow* via K-means culstering. The above plots show the latency distribution of the clusters in that order. The violin plot shows the distribution, mean latency, and median latency for each hardware. The *fast*, *medium*, and *slow* clusters have a mean latency of 50mns, 115ms, and 235ms respectively. The Venn diagram shows the CPUs that constitute each cluster.

50 ms, 115ms, and 235 ms, respectively. To provide a visualisation of the latency measurements (beyond just a statistic like mean), we draw violin plots of the latency measurements of each device in Figure 4. The violin plots show the probability density of latency across networks (along with median and the inter-quartile range). Both the slow and medium clusters have individually similar distributions indicating homogeneity, while the fast cluster shows some diversity, denoting two sub-clusters. In all cases, we observed that distributions are wide (note the log scale on the y-axis) and distinctly bimodal.

Mapping CPUs to device clusters. We now analyse the properties of the CPUs in each of the three clusters. In Figure 4, we also draw the Venn diagram to specify the CPUs in hardware devices belonging to each cluster. We observe that there exist some overlaps, *i.e.*, certain CPUs belong to multiple clusters. For instance, devices in both medium and fast clusters have Cortex-A53 and Cortex-A72 CPUs. Similarly, there are devices in all the three clusters that use a Kryo 280 CPU. However, in most cases (80 out of 105 devices) the CPU uniquely determines the device cluster. The average frequency for the fast, medium, and slow clusters are 2.67, 2.1, and 1.9 GHz, respectively, while the mean DRAM capacities are 6, 3, 2 GB respectively. Thus, we have two observations: one, the CPU often maps a device to a specific cluster, and two, the specifications of CPUs across clusters are predictably varying. This creates the expectation that we can estimate the latency of a network based on specifications of the CPU.

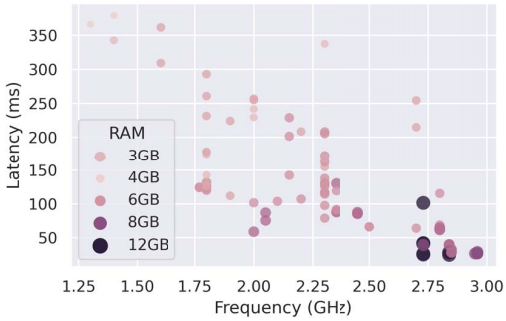

Fig. 5. Latency vs. Frequency for various DRAM sizes for MobileNetV2 across 105 devices. The hue in the graph represents the DRAM size. There's a decreasing trend of average inference latency with increase in frequency. But for a given frequency, there is significant variability in latency, even for the same DRAM size.

Relating latency to CPU specification. We evaluate the

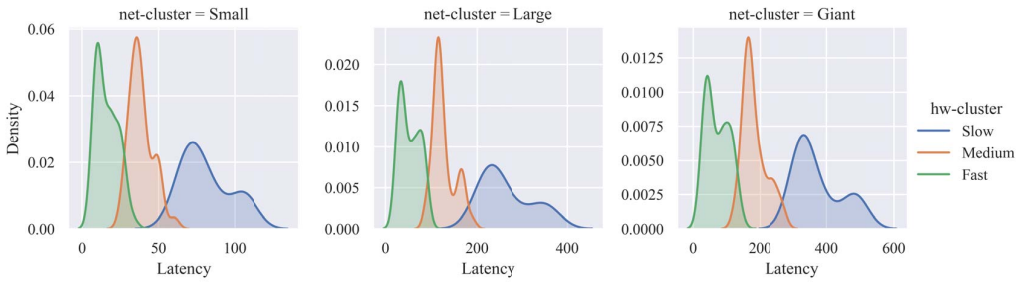

Fig. 6. Distribution of latencies of devices in fast, medium and slow hardware clusters on small, large and giant network clusters. There are two key observations: (i) Latency distribution of hardware clusters have similar patterns across network clusters, and (2) The overlapping latency distribution of hardware clusters indicate that predicting the latency for a hardware based on its cluster is not possible.

above expectation by visualising in Figure 5 the latency measured for MobileNetV2 [8] on all devices. The x-axis denotes frequency and the hue captures the DRAM capacity (a darker color denotes a larger DRAM capacity). Although we can observe a decreasing trend in latency with increase in frequency and DRAM size, there is still a significant variation. For instance, devices that run at 1.8 GHz and have 3GB DRAM capacity show *over 2.5x variability in latency*, ranging from 120 to 300ms. Clearly, this range is too broad to enable accurate workload characterization. Hence, simple hardware features like the core frequency and main memory size seem insufficient in predicting latency. We analyse this more formally and reach the same conclusion in the next section, by measuring the accuracy of a machine learning model trained with these features.

There are many other micro-architectural features such as out-of-order processing, super-scalar processing, cache size, and prefetching, which could affect latency. Analyzing the right set of these features and quantifying their predictive accuracy is a potentially interesting research direction. However, we surmise that such detailed characterization is not readily available to software developers looking to deploy DNNs on to a wide range of devices. We thus take an alternative approach of characterizing a device by a small number of latency measurements, as will be discussed in the next section.

Controlling for both device and network clusters. Just as we clustered the devices, we can similarly cluster the networks where each network is represented by a 105-dimensional vector given by the latency on all devices. We similarly found that k-means clustering provided a good trade-off at k=3 with three clusters which we name as *small*, *large*, and *giant* (due to a trend in number of FLOPs). Given this clustering, we can ask: "How discriminative of latency are device and network clusters". In other words, how distinct are the latency distributions when we control both for the device and network clusters. We show this with the help of faceted density estimate plots in Figure 6. In the case of all three network clusters – small, large, and giant – the distributions across device clusters are overlapping. Thus, even if we pick a specific cluster for both the device and network, the measured latency distributions are not distinct from each other.

In summary, our exploratory data analysis on the large set of measurements reveals the qualitative conclusion that measured latency shows a large distribution which cannot be satisfactorily explained by simple hardware specifications or network classes.

III. Towards Generalizable Cost Models

In this section, we present our main contribution on how to represent a given network and hardware, and then discuss how to use such representations to learn a latency model that generalizes across hardware and networks.

A. Overview

What would it take for a cost model to generalize across hardware and networks? At the outset, we need *representations* of both the network and the hardware. That is, given a DNN we need to map it on to a vector, say N, that represents the key features of that DNN. Similarly, each hardware needs to be mapped to a vector of key features, say H. Then a machine learning model, say M, could be trained to take as inputs N, H and estimate the latency l of that network on that device. Once such a model is trained, it can be used to estimate the latency for different network-hardware pairs. A model is said to generalize well if it accurately estimates the latency on networks and hardware devices that are not present in its training set. The natural question then is to identify right choice for the representations N and H. A representation must capture all features that are relevant to estimate the latency, but avoid extraneous features that can slow down data collection or increase inaccuracy in the training process. The representation for a network is more straightforward and discussed first. Subsequently, we discuss different choices for the hardware representation.

B. Network Representation

A DNN can be viewed as a graph with nodes describing operators and edges describing the flow of data between the operators. Example operators include convolution, fully-connected neurons, recurrent neurons, pooling, activation and skip connections. Often, the graphs are linear wherein operators are stacked sequentially in what are called *layers*. Each

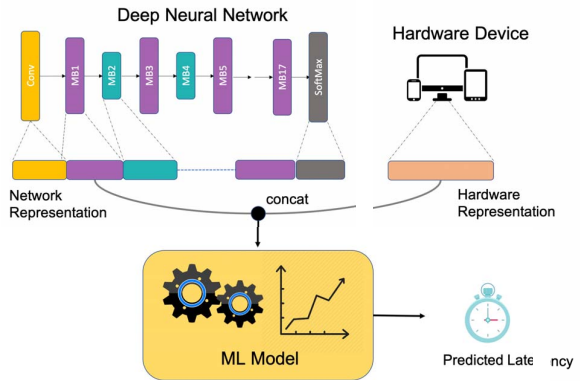

Fig. 7. This diagram shows our proposed cost model that takes in the representation of DNN and hardware as input to predict the latency of that DNN on that hardware. We represent each DNN by recursively encoding the layer identifier (one-hot) and its parameters. Each hardware is represented using a vector containing measured inference latencies on the signature set. We use XGBoost to train our cost model on inference latency data.

operator or layer has specific parameters, which characterize how it operates on the input. For instance, a convolution operator has parameters such as stride, padding, and kernel size. Thus, we can represent a DNN layer-wise: For each layer, we can represent the input and output sizes as is, the operator as a one-hot encoded vector, and its parameters as a sequence of numbers. Such representations across layers can be concatenated to create a single vector for the network. The length of such a representation would depend upon the number of layers in the network and thus would vary across networks. To enable that such representations can be processed by a larger class of machine learning models, we apply the standard procedure called masking, whereby we pad each representation by dummy values to match the size of the longest representation. Thus, our representation of a network is a concatenation of layer-wise representations as illustrated in Figure 7. We note that, though this proposal is only one of several representations of DNNs, there is neither much ambiguity or choice in the features to capture. On the other hand, the representation of the hardware is more open to choice, as we discuss next.

C. Hardware Representation

The hardware representation should map a given device on to a vector that captures all hardware characteristics required to predict latency on different networks. A first solution is to utilize essential hardware specifications, such as core frequency, main memory size and CPU family. Specifically, we choose to represent a device by three components: a one-hot vector representing the CPU model (*eg.* Cortex-A53 or Kryo 485, see Figure 3 for the full list in our dataset), an integer corresponding to the CPU frequency, and an integer corresponding to the main memory size. As we discussed earlier, while more features such as cache size or pipeline depth can be added to the representation, they are not readily available to software developers. Further, the CPU model itself may represent these more detailed features such as pipeline depth. For instance, we would not expect that two devices with the same make (say Cortex-A53) and the same frequency would differ in specific micro-architectural features.

In order to measure the effectiveness of such a hardware representation in learning latency models, we train a machine learning model of the form shown in Figure 7, and evaluate its accuracy. We use XGBoost, which is an efficient model based on parallel tree-boosting algorithm, as the model. The XGBoost-based models are seen to work well in practice especially in problems where training data-set is not particularly large. Further, in our experiments XGBoost outperformed many other models, including an LSTM-encoder followed by a fully-connected neural network, a random-forest model, and k-nearest neighbour models. First, we represent the 118 networks and the 105 hardware devices as discussed. We then split the devices randomly into train (70%) and test (30%) sets. The XGBoost model was trained on the train set using the hyperparameters *gbtree* booster with *lr = 0.1, n_estimators=100* and *max_depth = 3*. The trained XGBoost model then was used to estimate the latency for each of the points in the test-set. The actual and predicted latency for each of the inputs are shown in a scatter plot in Figure 8. The hue of the points denotes the CPU frequency. We observe that the model is unable to learn accurately as the points are spread out far from the ideal $y = x$ line. The coefficient of determination, also popularly called R^2 in this case was 0.13, which indicates a poor predictive accuracy.

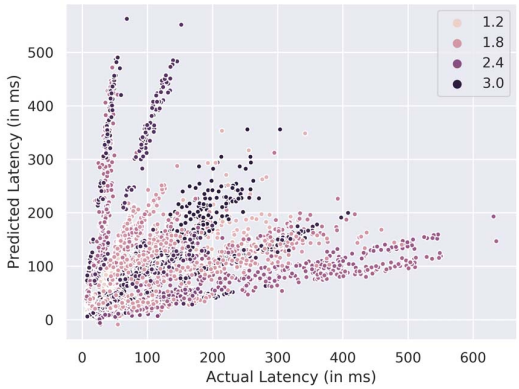

Fig. 8. Actual runtime versus predicted runtime of a cost model trained using static parameters of a device as hardware representation. R^2 Correlation value is 0.13.

The above experiment reveals that the proposed hardware representation is inadequate. As an alternative, we pose the following question: *"Can the latency of a chosen small set of networks on a hardware platform be informative enough to represent that hardware?"*. In other words, a device is represented not by its hardware specifications, but by measuring latency of a small set of chosen networks. We call this chosen set of networks the *signature set*. Note that such

a representation is convenient for software developers as it can be obtained by executing the signature set on a given device and measuring its latency. There are now two essential questions. First, how big does the signature set need to be? Second, how do we choose the networks that constitute the signature set? We answer these two questions by proposing three approaches in the remainder of this section. These approaches are evaluated in the next section.

1) Random Sampling (RS): As the first approach, we consider a simple random sampling (RS) technique. Given a user-defined number of networks, we choose signature set by uniformly sampling from the set of all networks. This sampling requires a definition of the *population* of networks, which must include a large diversity of networks that are expected to be seen during latency estimation. For instance in our case, the population includes the 118 networks comprising both popular networks and generated networks. Since there is no prior in selecting which networks form the signature set, the performance of any trained cost model may vary across samples.

Unlike the RS approach, the next two proposed methods choose a signature set by maximizing a metric of gain. The two metrics of gain used are (i) Mutual Information, and (ii) Spearman Correlation Coefficient.

2) Mutual Information Selection (MIS): Each network in the signature set provides information on latency of a device on the specific operators in that network. This suggests that including two networks which have very similar latency patterns across hardware devices is to be avoided. In other words, we want to include networks in signature set which contribute unique characterizations of the hardware device. Based on this intuition, we propose to use *mutual information* or equivalently information gain to guide the choice of signature set. Mutual information between a pair of random variables captures the amount of information obtained about one random variable by observing the other random variable. More formally, it is given by the following relation between joint and marginal probability distributions of random variables X and Y:

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p_{(X,Y)}(x,y) \log \left(\frac{p_{(X,Y)}(x,y)}{p_X(x).p_Y(y)} \right)$$

Given this definition, a good choice of signature set is a subset of networks that maximize the mutual information with the entire set of networks. Note that we treat the networks in signature set and outside it as two random variables whose samples are given by the latency measurements on the devices. Finding an optimal subset suffers from combinatorial explosion. As an alternative, we follow a greedy iterative approach, wherein in each iteration we add a single network into signature set. This network is chosen such that its addition maximizes the mutual information between the signature set and other networks. It is known that the function that maps mutual information across signature set is a submodular function and consequently a greedy algorithm works well with a constant approximation factor [18]. We detail the iterative process in Algorithm 1, which takes as input the $dataset$ matrix of latency measurements where n networks are arranged in rows and its latencies on h hardware devices are arranged in columns. The initial network is chosen randomly and all subsequent networks are chosen to maximize the greedy mutual information objective. The number of networks, m, to be added to the signature set is a user-defined variable.

Algorithm 1: Mutual Information Selection (MIS)

Input : $dataset$, n×h matrix containing inference latencies of n networks on h hardware
Input : m, Number of networks to be selected
$subset \leftarrow \{$`initialChoice`(network)$\}$
for $k = 1$ **to** $m - 1$ **do**
  $max\_info \leftarrow 0$
  for $j = 1$ **to** n **do**
    $subsetTemp \leftarrow subset \cup j$
    $newV \leftarrow dataset[j]$ `// h sized vector`
    $subsetMat \leftarrow dataset[subsetTemp]$
    $info \leftarrow$ `ComputeMI` $(subsetMat, newV)$
    $index \leftarrow (info \geq max\_info)\ ?\ j : index$
    $max\_info \leftarrow$ `max` (*info, max_info*)
  end
  $subset \leftarrow subset \cup index$
end

3) Spearman Correlation Coefficient Selection (SCCS): While the Mutual Information metric captures conditional probability between pairs of random variables, we may want to abstract some of the detail in the measurements. For instance, instead of predicting the exact latency of different networks for a given hardware, we may only be concerned with sorting the networks in terms of latency. This is particularly relevant in finding an efficient network for a given hardware, as is done in NAS. We thus propose to look at the Spearman correlation coefficient as an alternative metric for choosing the signature set. This coefficient is defined as the Pearson correlation coefficient on the rank of the two random variables, *i.e.*, instead of computing the Pearson correlation on two variables X and Y, we instead compute it on the ranks of the variables.

Unlike in the case of Mutual Information, the Spearman Correlation Coefficient can only be computed for networks pair-wise. We thus follow a modified iterative greedy heuristic to choose signature set. First, we compute the coefficient for each pair of networks, and then choose the network which has the highest number of coefficients greater than a threshold γ (typically close to 1). We then remove from the set of all networks, all those networks which have a correlation coefficient with the chosen network greater than γ. This removal signals that those networks which are highly correlated to the chosen network can be removed from further consideration. In the remaining set of networks, this procedure is repeated to add each time one additional network to signature set. We detail this procedure in Algorithm 2, which takes as input a

square matrix, ρ, where each cell represents the correlation coefficient for a given pair of networks.

Algorithm 2: Spearman Correlation Coefficient Selection (SCCS)

Input : ρ, n×n matrix containing Spearman Correlation between all pairs of networks across all hardware
Input : m, Number of networks to be selected
Input : γ, Correlation Threshold
$subset \leftarrow \emptyset$
for $k = 1$ **to** m **do**
  $index \leftarrow arg\,max_i \sum_{j=1}^{n} \rho[i][j] \geq \gamma$
  $subset \leftarrow subset \cup index$
  $highCorr \leftarrow \emptyset$
  for $j = 1$ **to** n **do**
    if $\rho[index][j] \geq t$ **then**
      $highCorr \leftarrow related \cup j$
    end
  end
  Delete $highCorr$ networks from ρ
end

In summary, we discussed the network and hardware representations. The network representation is a representation of layer-wise features. The hardware representation given by top-level specifications did not produce accurate cost models. We thus consider the usage of a signature set to represent a device, and propose three ways to select the signature set.

IV. Results

In this section, we first present the methodology adopted in our experiments. We then present results which evaluate the cost models that are learnt with the different hardware representations.

A. Experimental Methodology

In all our experiments we use XGBoost as the ML regression model of choice to build the cost model. To evaluate the generalizability of our learnt models, we split the devices into train (70%) and test (30%) sets. The two sets contain the latency of all the 118 networks for every hardware in that set. Only the hardware devices in the training set, participate in choosing the signature set of networks for the hardware representation. Once the signature set networks are chosen, their latency on all the hardware devices in the train and the test set are discarded. We then train the XGBoost model with the remainder of the training set data. We use the root mean square error (RMSE) loss function for optimizing the learning model. The same hyperparameters described in Section III-C are used. Once the model is trained, we evaluate it on the test set with the co-efficient of determination (R^2) as the metric.

B. Comparison of Methods to select signature set

As the first set of experiments, we choose 10 as the size of signature set and compare the three methods of choosing the signature set. The accuracy of the models trained with three different *signature sets* are illustrated in Figure 9. We plot the actual-vs-predicted latency for all network-hardware pairs in the test set. The points are closer to the $y = x$ line denoting higher accuracy (in contrast with Figure 8). Quantitatively, the accuracy of the model is captured by high R^2 values: 0.9125, 0.944, and 0.943 for RS, MIS, and SCCS, respectively. Notice that the plots in Figure 9 also qualify the generalization of the models, since the plotted values are on the test set which have hardware devices that are unseen by the model during training. Amongst the three approaches to choose signature set, MIS and SCCS perform better with higher R^2 values. However, we find that even the naïve choice of signature set based on random sampling performs competitively. This illustrates the effectiveness of our approach of representing a device by latency measurements on few (in this case 10) networks.

There are two further questions. First, are there any challenges with random selection of signature set - in particular is the method robust with no poorly performing outliers? Second, what is a good choice for the size of signature set?

C. Variation across randomly chosen signature set

We now study the robustness of the randomly chosen signature set. We randomly sample 100 different *signature sets* each of size 10, and use them to compute hardware representations. For each representation, we train a different model with XGBoost. Figure 10 shows the test R^2 score for each such trained model. From the plot it is clear that models learnt with randomly chosen signature set hardware representations perform on average competitively. The average R^2 score for RS is 0.93, compared to 0.944, and 0.943 with MIS and SCCS, respectively. However, there are samples where the model performs relatively worse: a low R^2 score of 0.875 for 2 of the models. These outliers, though infrequent, highlight the challenge with random sampling, wherein they may generate representations leading to poor models. Thus, we recommend the deterministic methods MIS or SCCS for selecting signature set.

D. Evaluation of different sizes of signature set

We now evaluate the performance of models when using different sizes of signature set. Clearly, a larger signature set captures more detailed features of a hardware and may be expected to increase accuracy of the cost model. However a larger signature set implies higher cost in collecting the latency measurements. We explore this trade-off for the three methods for choosing signature set. Figure 11 plots the R^2 metric for the trained models for each choice of number of networks in signature set. For the random sampling method, we report the R^2 values by averaging over 100 samples.

There is a general trend of improved accuracy on increasing the size of signature set. In the case of random sampling, we find that there is a consistent improvement with the rise in

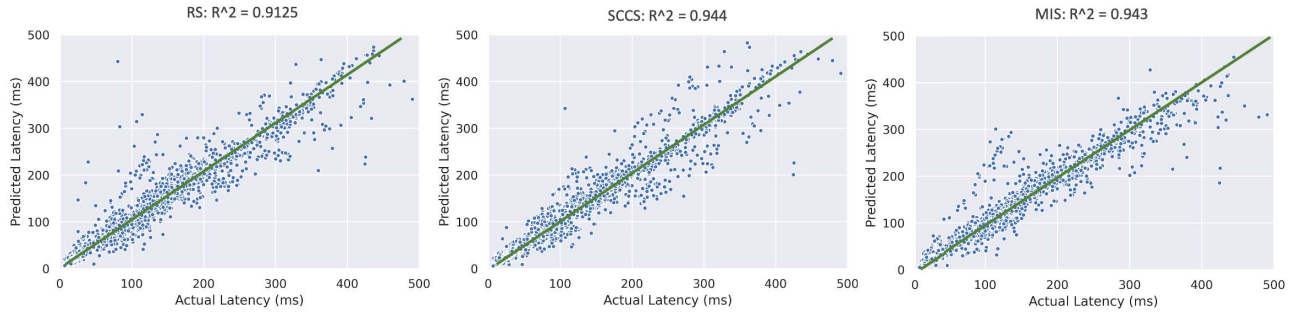

Fig. 9. Actual vs Predicted runtimes (in ms) for our hardware cost models learnt using the proposed hardware representations under different sampling techniques. *y=x is the ideal regression curve expected.*

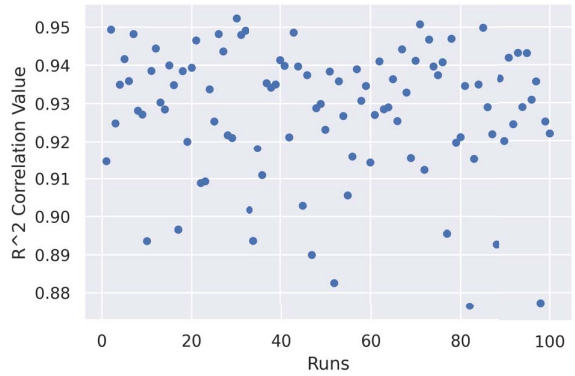

Fig. 10. R^2 value for 100 different randomly chosen *signature sets*. They perform competitively against MIS and SCCS. However, there are some outliers that may generate representations leading to poor models.

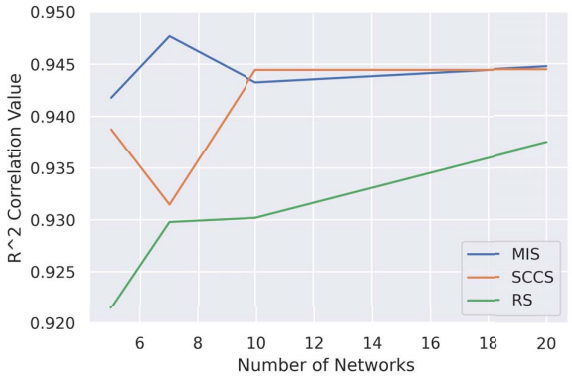

Fig. 11. Accuracy scaling with increase in the number of networks in signature set. A larger signature set captures detailed features of hardware. However, it implies higher characterisation cost.

the number of networks, even to sizes of over 20. Thus, if we could afford a larger set of measurements, then random sampling provides a simpler method to select signature set. However, we recall again the caution that some of these random samples may be outliers and lead to inaccurate models. For MIS and SCCS the R^2 is around 0.94 even for small *signature sets*. Based on these results, we identify that a signature set of sizes 5-10 are good choices when using these deterministic selection methods, amounting to a sampling ratio of 4-8% from the total set of networks (118 in total). Selecting networks beyond this fraction does not increase accuracy indicating a saturation in mutual information gain in the case of MIS, or high pair-wise correlation in the case of SCCS.

E. Generalizability between hardware clusters

So far, the results have indicated that signature set is an effective way of representing hardware for training generalizable cost models. We now experiment to identify limits to this generalizability. In particular, we consider adversarial splits of the hardware devices amongst train and test sets. Instead of splitting the devices randomly between train and test sets, we choose them based on the clustering presented in Figure 4 into fast, medium, and slow devices. As the violin plots of Figure 4 show, devices across these clusters differ significantly in their latency characteristics. We choose devices from two clusters as the training set, and those from the third as the test set. The size of the signature set is set to 10. For each of these settings we report the R^2 values on the trained models for signature set chosen according to the three methods - RS, MIS, and SCCS in Table I. As an example, the R^2 value of 0.912 in the first cell is for the case of random sampling of signature set, the model trained on the medium and slow clusters, and tested on the fast cluster.

From the results we make two observations. Firstly, the models generalize relatively better when medium and slow clusters are used as test sets. This suggests that devices in the fast cluster generate more generalizable cost models. Conversely, when the fast cluster is used as the test test, models perform poorly indicating that the devices in the medium and slow cluster do not learn cost models that generalize to the fast cluster. Perhaps, the devices in the fast cluster have micro-architectural features quite different from those in the other two clusters. Thus, when in the devices chosen in the training set, diversity must be maintained. Secondly, we highlight again the surprising generalization that the cost models show across hardware. With measurements on CPUs of fast and

TABLE I
TEST SET R^2 SCORE FOR A COSTMODEL TRAINED ON TWO HARDWARE CLUSTERS AND TESTED ON THE THIRD CLUSTER

Subsetting Technique	Coefficient of determination (R^2)		
	Fast	Medium	Slow
Random Sampling	0.912	0.964	0.975
Mutual Information	0.916	0.973	0.967
Spearman Correlation	0.949	0.976	0.97

medium mobile devices and representations of hardware on 10 randomly sampled networks, the trained models predict latency on slow mobile devices with R^2 over 0.97. This result reaffirms our confidence that the presented methodology is effective in learning generalizable cost models.

V. PROPOSED COST MODEL IN PRACTICE: COLLABORATIVE WORKLOAD CHARACTERISATION

In this section, we discuss a practical setting wherein generalizable cost models are used to collaboratively characterize networks on different hardware. With a simulation on the real data-set we quantify the benefits of such collaboration.

Our method proposed so far has two requirements: (a) representation of each hardware in terms of the measured latency on a signature set, and (b) a training set of latency measurements on a diverse set of devices. Our results show that the signature set can be fairly small, but the training set needs representation of a diverse set of devices. Thus, the requirement of the training set is still a bottleneck. Principally there are two approaches to generate a training-set. One, collect latency measurements on a common but large set of different networks for a limited set of devices. Two, collect a small number of measurements on a very large set of diverse devices. The question then is: "For the latter case to be more effective, how small can the number of measurements be for each device and how large must be the set of devices?" Or differently stated, in the latter collaborative case, how many devices should collaborate and what should be the quantum of contribution of each device for training accurate cost models.

A. Simulation methodology

To study the collaborative case, we run a simulation with our measured values. The simulation is iterative and is as below:

- First we choose the signature set of size 10 using MIS on all networks (from our collected list of networks).
- In each iteration, a new hardware platform (from our collected list of devices) is added. Such a hardware platform contributes its representation as given by latency on signature set. In addition, each hardware contributes latency on a small percentage (10-30%) of randomly chosen networks (from our collected list of networks).
- At each iteration, a XGBoost model is trained to predict latency where the training set comprises all latency measurements contributed by previously chosen hardware devices. We then report the model's average R^2 when evaluated on *all* networks (in our collected list of networks) for the hardware devices added till then.
- This process continues for 50 such iterations.

This simulation thus studies the evolution of the quality of a model trained incrementally as one new hardware is added, wherein the contribution from each hardware is the measurements on the signature set and a small percentage of the networks as training data.

B. Evaluation

First we study how the models learnt collaboratively evolve as more devices are added. Figure 12 shows the average R^2 of the learnt collaborative model as each device is added. In addition, we perform experiments for varying percentages of network contributions for each device, from 10-30%. Note that the average R^2 is taken across added devices on all networks (well beyond the training set). Clearly, models become more accurate as devices continue to be added. We find it surprising that even with 10 devices the trained models have R^2 values greater than 0.9 in each case. However, if we are interested in highly accurate models with R^2 greater than 0.95, more than 40 devices would be needed. We also observe that when each device contributes as low as 10% of randomly chosen networks as training set, we can still learn accurate models.

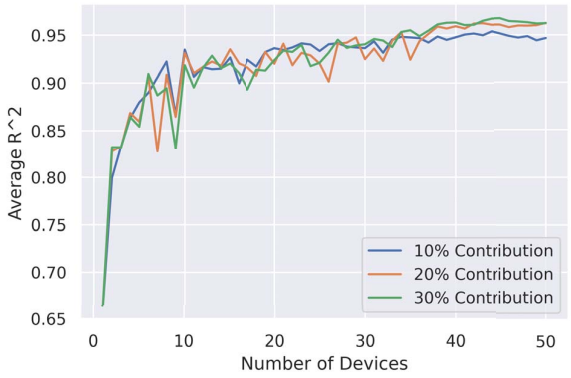

Fig. 12. Average R^2 score of the collaborative cost model with increase in the number of devices with each device contributing 10-30% of networks as training data.

C. Collaborative vs isolated training of the cost model

The above experiment quantified the evolution of accuracy of the collaborative model. However, to fully value the collaborative process we compare it with the case of training a cost model for a hardware, separately without collaboration. We illustrate this for the Redmi Note 5 Pro device, that uses a Kryo 260 Gold CPU.

First, we train a latency cost model for Redmi Note 5 Pro. Instead of training a single model, we train an entire sequence of models varying the number of networks in the training set from 1 to the entire set of 118 networks. Each such model is evaluated on a test-set of all 118 networks, in terms of the R^2

score. The evolution of these R^2 scores as more networks are added is shown in Figure 13. Clearly accuracy improves with more networks, but with marked regions of slow and rapid progress.

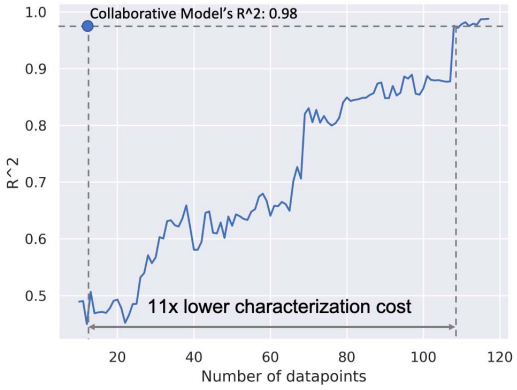

Fig. 13. R^2 score for the collaborative model vs. individual model for Redmi Note 5 Pro. The accuracy of the single model improves as more networks are added. However, in the case of collaborative cost model it has an R^2 value of 0.98 on the test set of all networks for contributing just 10 measurements (11x lower) making a compelling case for collaboration.

We contrast this with the case of collaborative cost model. We consider 50 randomly chosen devices, of which Redmi Note 5 Pro is one, with each device contributing only 10 measurements on the signature set and 10 measurements on other randomly chosen networks. In this case, the learnt cost model has a R^2 value of 0.98 on the test set of all networks on the Redmi Note 5 Pro device. Thus, with just 10 measurements from each device, we obtain an accuracy on the Redmi Note 5 Pro device which matches that when trained with more than 100 networks (see Figure 13).

This experiment makes a compelling case for collaboration. Multiple parties can share latency measurements and network properties to a shared repository along with latency measurements on a commonly agreed signature set. If such a collaboration can be enabled, then joint cost models can be learnt whose accuracy would otherwise require an order of magnitude more number of measurements if done in isolation for that device.

VI. Related Work

There have been many prior works that use a cost model to predict the execution of a DNN for a given hardware. The different efforts can be categorized into two broad themes, based on the context of the modelling: *domain specific compilers* and *neural architecture search*.

Domain specific compilers for Deep Learning use latency cost models to guide the exploration of finding an optimal mapping of a DNN for a given hardware platform [3], [4]. One such compiler, TVM, proposes the use of XGBoost [19] and Tree-GRU based learning models to predict the runtime of a DNN operator for a given hardware target. It employs a novel, transferable input representation of high level programs based on its Abstract Syntax Tree (AST) to enable generalization across a wide range of DNN operators. Halide is another compiler effort for image processing [4] that uses a simple MLP based cost-model guided by a carefully chosen set of large number of program features for predicting the execution-time of a program schedule. A recent effort from Google [5] develops a cost-model to accurately estimate the execution-time of an ML model running on Tensor Processing Units (TPU). They use GraphSage, a graph library to extract features or embeddings from a DNN, followed by a feedforward layer to estimate the execution time of the DNN on TPU. All these efforts show strong generalizability across a wide range of DNNs and its operators. However, the cost models are often trained online, along with the search process, for a specific hardware platform. Thus these methods are not designed or tested for generalizability across devices. In contrast, our work focuses on representations of different devices and learning cost models that generalize across a wide variety of mobile devices.

Neural Architecture Search (NAS) is an increasingly popular technique of employing novel search strategies to design DNNs within a large space of DNN operators and topologies. Often, hardware parameters such as latency and energy efficiency are used as feedback to guide the search process. In order to reduce the cost of obtaining these parameters from the hardware, cost models are employed. For instance, ProxylessNAS [2] and Once-for-All [20] employ a simple MLP cost model to predict the latency of DNNs on mobile devices. Similarly, ChamNet [21] creates a Look Up Table (LUT) of common DNN operator latencies for Samsung S8 and uses an additive model to find the latency of any DNN that is composed of these common operators. Similar to compiler efforts, these cost models for NAS have to be trained for every device separately. Again, our primary focus is to demonstrate generalization across a large set of devices.

The methods proposed in our work are orthogonal to the context in which the cost models are used: during compilation or neural architecture search. Further, our work can be combined with other representations of DNNs such as the graph-based deep representations as in [5].

VII. Conclusion

Characterizing latency of DNNs on hardware devices is challenging due to both network and device diversity. We proposed a novel and easy-to-obtain representation of devices given by the measured latency on a small signature set of networks. We showed that with a careful choice of the signature set and the machine learning model, we can learn accurate models that generalize across networks and hardware. We demonstrated these results on measurements collected in the real world on a large diversity of mobile devices. We also discussed how collaborative workload modelling is significantly more efficient in comparison to learning cost models separately for each device. Our work thus recommends

building a central repository, where network representations and latency measurements across devices are used to train a continually learning global cost model. Such a global cost model would significantly reduce the computational and environmental overhead in characterizing and fine-tuning DNNs, both during Neural Architecture Search and domain-specific compilation. These results would be strengthened by extending them to desktop- and server-grade devices.

REFERENCES

[1] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[2] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[3] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[4] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[5] S. Kaufman, P. M. Phothilimthana, and M. Burrows, "Learned TPU Cost Model for XLA Tensor Programs," in *Proceedings of the Workshop on ML for Systems at NeurIPS 2019*, 2019, pp. 1–6.

[6] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[10] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[11] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, and D. Marculescu, "Single-path nas: Designing hardware-efficient convnets in less than 4 hours."

[12] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1314–1324.

[13] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[15] "Tensorflow Lite." [Online]. Available: https://www.tensorflow.org/lite

[16] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," https://github.com/onnx/onnx, 2019.

[17] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[18] A. Krause, A. Singh, and C. Guestrin, "Near-optimal sensor placements in gaussian processes: Theory, efficient algorithms and empirical studies," *Journal of Machine Learning Research*, vol. 9, no. Feb, pp. 235–284, 2008.

[19] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[20] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[21] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia *et al.*, "Chamnet: Towards efficient network design through platform-aware model adaptation," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2019, pp. 11 398–11 407.