Cost aware inference with early exits

Vishal Keshav
vkeshav@cs.umass.edu

Berett Babrich
bbabrich@umass.edu

Paige Calisi
pcalisi@umass.edu

April 22, 2020

Abstract

In modern computing, there is an increased need for performing inference computation on-device due to privacy and efficiency concerns. For applications requiring continuous, non-critical predictions, the need for fast predictions outweighs the need for perfect accuracy. Existing solutions seek to reduce the required computation for a single round of inference, while ignoring the variability in inference runtime caused by the interfering background load. In this paper, we present a novel approach to dynamically adjust the model execution and provide a deterministic policy that strikes a right balance between accuracy and efficiency. The objective of our research is to reduce runtime performance variability to enhance continuous, realtime predictions.
Keywords: *edge computing; machine learning; dynamic inference, real-time prediction.*

1 Introduction

AI computing has moved out of the traditional cloud servers and is getting closer to where the data is being generated. Privacy and security concerns surrounding the ML applications have only fueled this shift, driving innovative solutions to be proposed to improve the on-device inference. Traditional use-cases from computer vision and NLP domain are increasingly being deployed on smartphones and other edge devices like IoT. These applications can be categorized into four quadrants (shown in fig.1). Applications falling into the realtime non-critical category allow for the flexibility to miss the ideal accuracy threshold, but the inference needs to be continuous and without any delay. In this paper, we focus on non-critical realtime applications such as human pose detection (10).

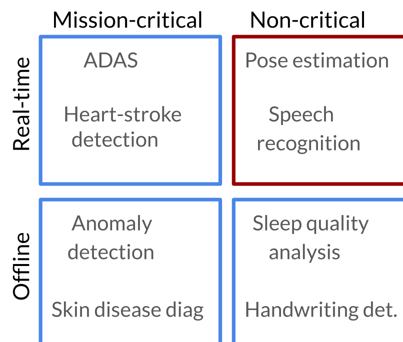

Figure 1: Application categorization.

One advantage, apart from improved security, that a realtime ML application gets from on-device inference is that of reduced latency, as this gets rid of server data transmission cost. However, other problems emerge that relate to power and performances. To resolve this issue, traditional research focuses on techniques that reduce the computation required to complete one round of prediction. These techniques include architectural improvements, quantization (9), and taking advantage of sparsity (3) (6) to discard computational blocks altogether. In the context of realtime non-critical applications, there still exists a problem that relates to variability in ML performance. In these applications, each round on inference is not guaranteed to provide a consistent execution runtime. This variability is mainly caused by processes such as other ML workloads and background system tasks. These processes can interfere with on-going real-time prediction and the resulting runtime can differ from one run to another. From the end-user perspective, this translates to jitter and lag in the application.

To resolve the performance variability problem, a new class of system and ML architectures is required that can enable dynamic real-time inference. The dynamics of the execution runtime could be controlled by factors such as background load, ML architecture characteristics and user's quality of service metrics (such as accuracy demands). Such a cost awareness may not only reduce the performance variability but also provide guarantees on runtime and accuracies across multiple runs.

Unfortunately, there does not exist a dynamic policy and a corresponding system to support such a dynamic inference for edge devices. Popular mobile-inference engines such as TFlite (5), Caffe (4), CoreML (2), and others primarily focus on improving the single execution cycle. In contrast to these existing systems, the focus of this paper is to improve the realtime application performance variability, guided by the appropriate QoS metric. In section 2, we discuss state-of-the-art approaches to resolve issues falling in this domain and discuss how our method is different from theirs. In section 3, we formulate the problem statement. In section 4, we describe a simple policy and propose a system to reduce performance variability. Section 5 details the implementation and finally section 6 provides some concluding remarks.

2 Related work

Most existing mobile-AI applications use various kinds of neural network architectures, such as convolutional neural networks, for computer vision tasks and recurrent neural networks for sequence-based tasks. These models are performance-intensive, and most of the compute demand is attributed to matrix multiplication. In the context of real-time prediction, while reducing the computation cost with minimal degradation in the accuracy is an important goal, reducing the computational cost for each run is equally important.

Non-critical realtime applications provide the flexibility to trade-off the computational cost with prediction accuracy. To do so, in the past, several approaches were proposed to do dynamic inference. In this section, we discuss two close approaches that describe a system to do partial execution.

2.1 Architectural design to support conditional computation

The idea of having conditional computation is based on terminating the prediction early with one of the computational branches. One such architecture is BranchyNet (12), where multiple exit points are augmented in an already existing neural network architecture. The location of the exit branches is chosen such that it strikes a good balance between computation cost and accuracy, if the computation is meant to be terminated early. The basis for terminating early can be based on several heuristics. For example, in the paper Conditional Deep Learning (11), the conditional computation was based on the toughness of the input (shown in fig.2). A different network was trained to take input and then predict the confidence through each of the exit points. Based on this confidence in the accuracy, one of the exit points is selected. This approach only caters to improve on the accuracy with the least computational cost. The first drawback of this approach is the unreliability of the confidence prediction system. The second drawback is that it disregards the computational constraints imposed from the system on which inference has to run (such as edge devices) and only focuses on maintaining the highest possible accuracy. In our approach, however, we provide a deterministic policy to decide the choice of an exit path and provision to the need for performance constrained edge devices and to the applications that can trade-off accuracy for efficiency. In summary, our objective is different, which is to reduce the runtime performance variability for realtime continuous prediction.

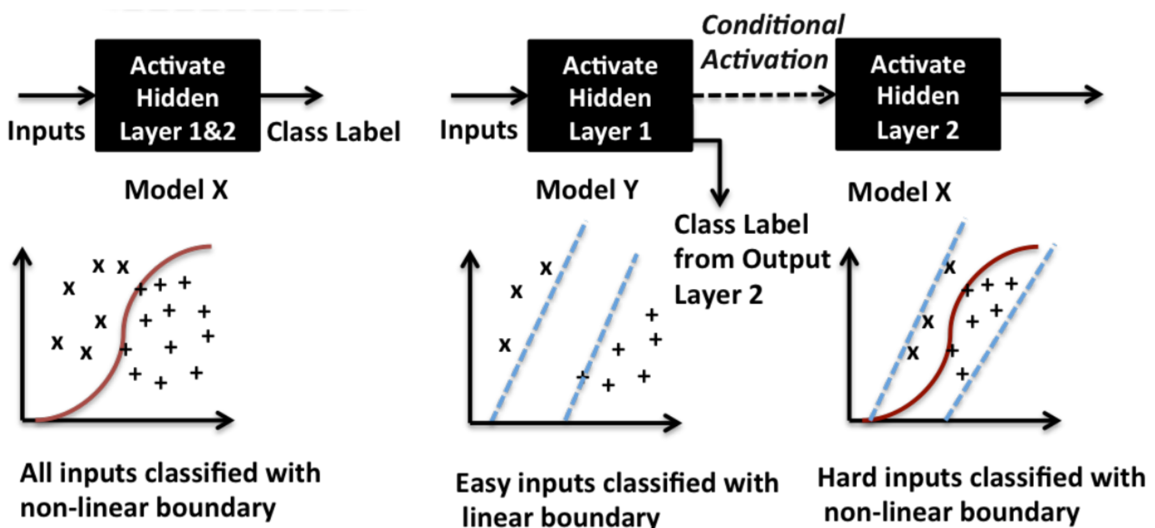

Figure 2: Early exit architectures and decision policy based on the input toughness.

2.2 Trainable throttling policies

A recent paper (7) generalizes the idea of early termination systems to build a throttlable architecture design. The paper proposes to use the gated modules which decide when to throttle an on-going computation. Through the gated mechanism, throttling can be done at multiple levels, such as at the layer level or at the neuron level. Furthermore, the proposed system is end to end trainable, such that policy can be adapted to the data and user requirement. The main drawback with this approach is that the training needs to be done in two phases, first to train the network for data path, and then train the network for gating mechanism. Unlike theirs, our approach does not involve the second phase of training, as the decision policy we propose is deterministic. The second drawback in their approach is that an already performing model cannot be utilized, instead, the architecture has to be constructed and trained from scratch. Our approach involves augmenting existing established architecture and utilize the hard work which has gone to design the neural network (for example MobileNet (8) for image classification).

3 Problem formulation

To concretely define our problem statement and evaluate our approach, we experimented with real-time image classification problem and used a widely adopted mobile-net family of architectures (8). The Mobile-net family of architectures provides two tunable hyperparameters (width multiplier and depth-multiplier), and with a different choice of the hyperparameter the architecture can manifest different accuracy-efficiency characteristics. Once the model is trained with a set of chosen tunable hyperparameters, the number of MAC (multiple add operation) and accuracy remains fixed. To demonstrate the runtime variability with these sets of architectures, we ran each model on an Android device (where we maintained the same background load environment, explained in the implementation section), collected 1000 runtime data, and plotted the runtime distribution. The accuracy-runtime characteristics of chosen mobile net architectures are shown in table 3 and the runtime distribution for each model is shown in fig.3.

MobileNet version	MACs (Millions)	Paramerers (Millions)	Top 1 accuracy	Lower bound runtime (ms)	Upper bound runtime (ms)
mobilenet v2 1.4 224	582	6.06	75.0	150	280
mobilenet v2 1.3 224	509	5.34	74.4	120	210
mobilenet v2 1.0 224	300	3.47	71.8	70	150
mobilenet v2 0.75 224	209	2.61	69.8	50	110
mobilenet v2 0.5 224	97	1.95	65.4	30	55
mobilenet v2 0.35 224	59	1.66	60.3	20	45

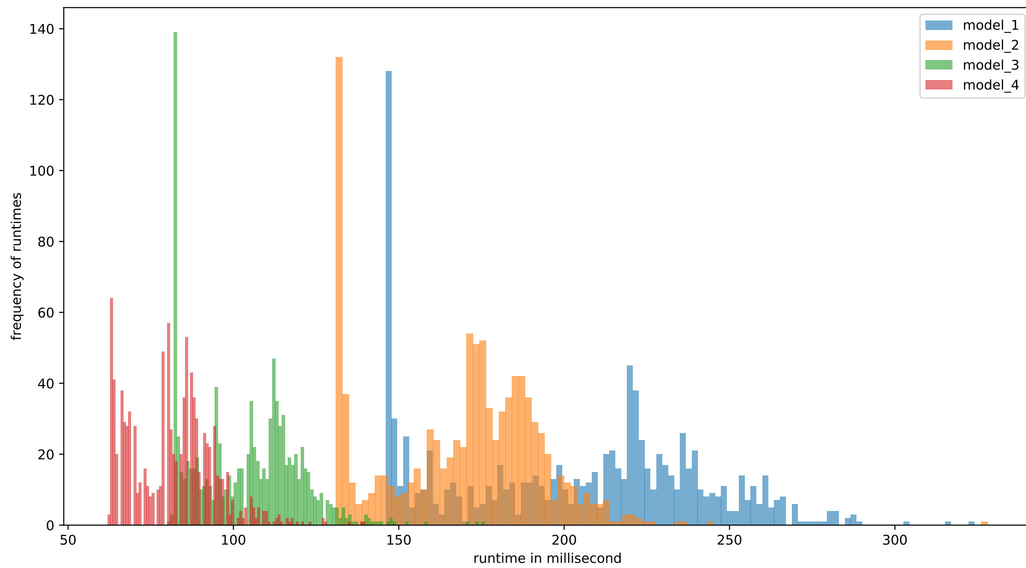

Figure 3: Runtime distribution with different versions of mobilenet, each model is run separately for 1000 runs.

During the multiple runs of inference, if only model 1 is chosen, then the accuracy will be highest, however, the runtime will suffer. On the other hand, if a lower model (such as model 4, 5, etc.) is selected, the runtime is guaranteed to be equal or less than the ideal runtime (defined as the lower runtime bound of model 1), but the accuracy will be low. To strike a good balance between the runtime and accuracy across different runs, we define a user quality of metric (denoted by α). The QoS metric α can vary from 0 to 1, where α = 1, means there is no concern for accuracy, but the runtime variability should be minimal. With α = 1, accuracy becomes the topmost priority. Anything in between penalizes both runtime variability and distance from an idea accuracy. Figure 4, demonstrates this case aptly.

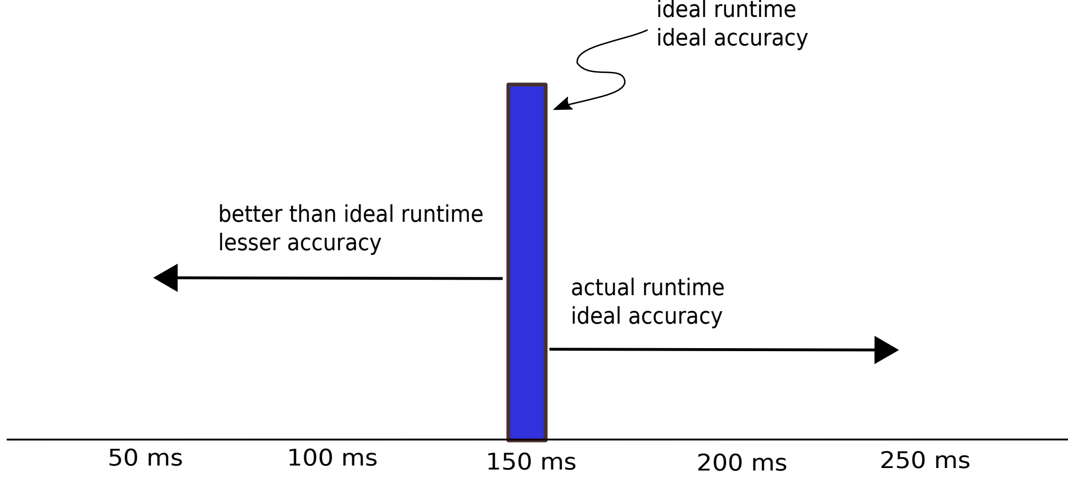

Figure 4: A visual aid to demonstrate the the trade-off between accuracy and runtime.

Formally, we define the runtime and accuracy penalty as follow:

$$R_p = max(0, r - \hat{r})^2 \quad (1)$$

where R_p is the runtime penalty, r is the actual inference runtime and $\hat{r}$ is the ideal runtime, which is the lowest runtime achieved from the best performing model/exit-path.

$$A_p = (\hat{a} - a) \quad (2)$$

where A_p is the accuracy penalty, $\hat{a}$ is the ideal accuracy (highest accuracy achieved from a model/exit-path) and a is the accuracy of the model chosen to run a single inference.

Total penalty across N runs is defined as:

$$T_p^{(N)} = \sum_{i=1}^{N} (\alpha * R_p^{(i)} + (1 - \alpha) * A_p^{(i)}) \quad (3)$$

where T_p denotes the total penalty across N inference runs, $R_p^{(i)}$ denotes the runtime penalty on the ith run, $A_p^{(i)}$ denotes the accuracy penalty on the ith run, and α denotes the QoS metric described above.

Our objective is to minimize the total penalty for a given user's quality of service metric α.

$$min_{k \in K} T_p^{(N)} \quad (4)$$

where $K = \{0, 1, 2, ..., M\}^N$ denotes the possible choice of M models (or exit paths) across N different inference run.

In the next section, we detail our approach to minimize the total penalty that takes into the account the model's accuracy-runtime characteristics and CPU load. The developed policy helps choose one of the models or one of the exit points (if a dynamic model like branchyNet with multiple exit points is used) dynamically.

4 Approach

Our approach to minimize the total penalty T_p depends on reliably predicting the model execution runtime for the different exit point. To do that, we track the instantaneous background CPU load by reading $/proc/stat$ file system at a regular interval (50 to 100 ms). Based on the CPU load and execution runtime data by executing the mobile net model for image classification 1000 times, we observed that the CPU load correlates with the inference runtime. As the background CPU load increases, the inference runtime increases as well. Furthermore, we are interested in the CPU load trend and not instantaneous loads (which has irregular peak behavior). To estimate the CPU load trend from instantaneous CPU loads, we discuss two methods:

4.1 Load trend estimation

(a) **Load prediction based on exponential moving average:** Let l_t denotes instantaneous CPU load recorded at time t, then exponential moving average load at time t (denoted by exp_t) is defined as $exp_t = \mu * l_t + (1 - \mu) * exp_{t-1}$. This weights the current load and all previously seen loads with exponentially decaying weights μ. This way of load estimation has the same shortcoming (abrupt peakiness) as that of the instantaneous load.

(b) **Load prediction based on window average:** The load prediction based on average of window of size W (denoted by avg_t) is give by $avg_t = sum_{t=t}^{t-W} l_t/W$. This load prediction method shows a better load trend by smoothing the previously observed load. The fig.5 shows the instantaneous load and load prediction achieved from the above-described techniques. It also demonstrates that window-based load estimation correlates better with the actual inference time.

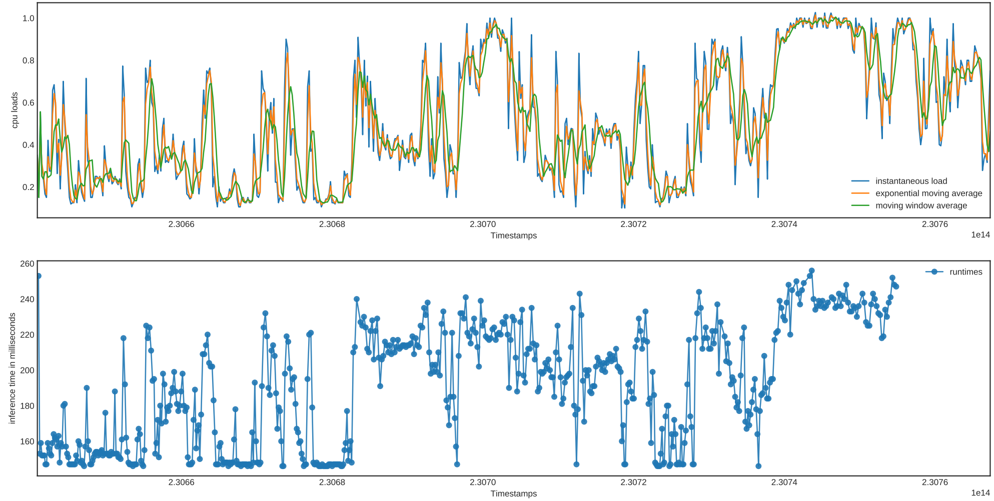

Figure 5: Load estimation and runtime correlation with CPU load with 1000 runs of MobileNet model for image classification.

To reliably estimate the runtime for each exit path, we scale the estimated load in the model runtime ranges. Specifically, if a model/exit path exhibit the runtime in the range L ms to U ms, then estimated runtime at CPU load l_t is given by:

$$\tilde{r_t} = (U - L) * avg_t \tag{5}$$

where $\tilde{r}$ denotes the predicted runtime at cpu load l_t.

Next, we provide the complete algorithm which is based on the minimization objective and runtime estimation system described above.

4.2 Exit path selection algorithm

In the plot shown in fig.6, we vary CPU load and capture the model selection with 6 mobile net model version described in the table3 with different values of alpha. The plot shows how different choices of alpha impact the model selection as CPU load varies. These selections are such that the total penalty is minimized.

Algorithm 1 Exit-path selection algorithm

1: **Input:** exp_t, **L**, **U**, **A** α
2: **Output:** Exit path index
3: **for** $i = 1, 2, \ldots |L|$ **do**
4: Set $\tilde{r^{(i)}} = L_i + (U_i - L_i) * exp_t$
5: **end for**
6: Set $\hat{r} = L_{|L|}$
7: Set $\hat{a} = A_{|L|}$
8: **for** $i = 1, 2, \ldots |L|$ **do**
9: Set $R_p^{(i)} = max(0, \tilde{r^{(i)}} - \hat{r})^2$
10: Set $A_p^{(i)} = \hat{a} - A_i$
11: **end for**
12: Normalize R_p and A_p
13: $T_p = \alpha * R_p + (1 - \alpha) * A_p$
14: $k = argmin\ T_p$
15: Return k

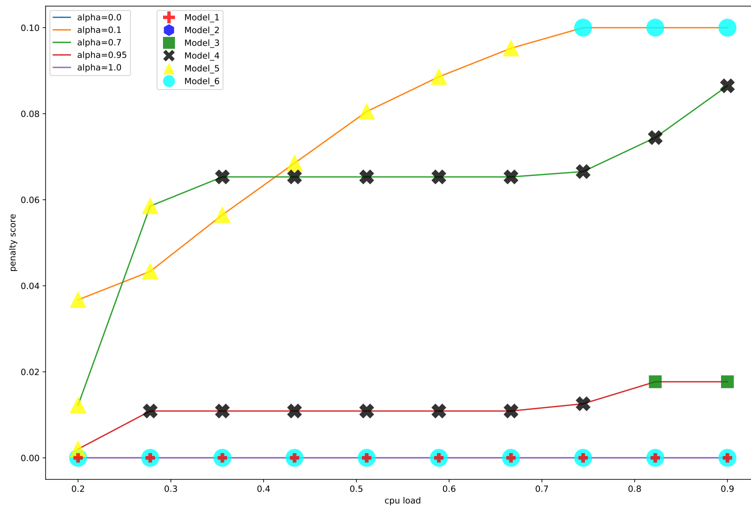

Figure 6: Penalty with varying alpha and cpu load.

5 Results

5.1 Results with MobileNet family of architectures

Our first set of experiments was conducted on the Mobilenet family of architectures (specified in the table). The exit path selection algorithm was modified to select one of the 6 mobile net architectures. The figure 7 shows the effectiveness of predicting model execution runtime by comparing it against the recorded inference runtimes. This demonstrates that the average CPU load is a good proxy to tell how the model will behave if ran with the random system load.

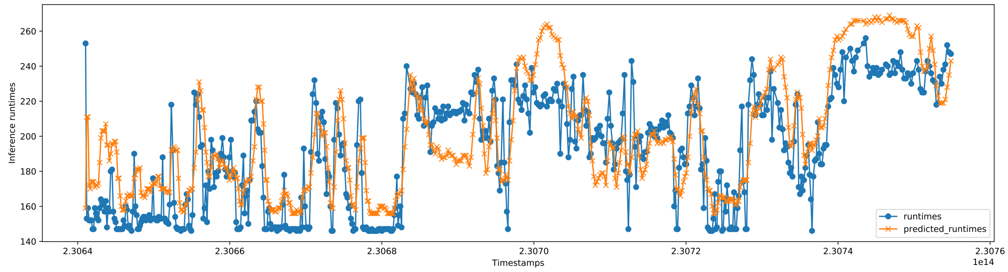

Figure 7: Estimated runtime and actual runtime.

Next, we list the average latency, variation in latency, and the achieved accuracy of each of the models individually, then compare it with our approach. Results are shown in the table5.1:

MobileNet version	Average runtime	Runtime variation	Average accuracy
mobilenet v2 1.4 224	202.02	37.94	75.0
mobilenet v2 1.3 224	169.26	19.05	74.4
mobilenet v2 1.0 224	103.77	1.32	71.8
mobilenet v2 0.75 224	81.49	0.0	69.8
mobilenet v2 0.5 224	40.326	0.0	65.4
mobilenet all ($\alpha = 0.5$)	**170.10**	**31.84**	**74.82**

The figure 8 shows the runtime distribution after applying the proposed algorithm. It can be observed that the runtimes have shifted towards 150 ms (which is the ideal runtime) mark.

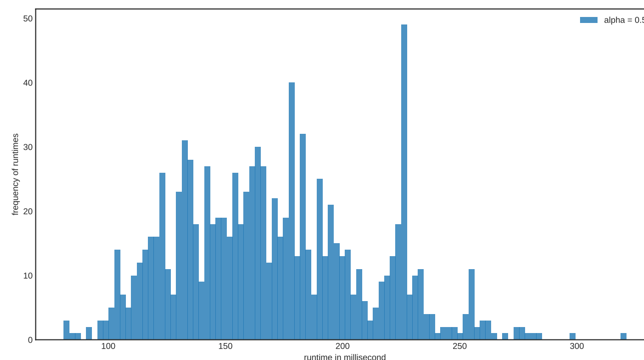

Figure 8: Runtime distribution with dynamic model selection at runtime.

5.2 Results with branch augmented MobileNet architectures

The second experiment relates to augmenting the mobile net with multiple exit points. Each of the exit point classifiers was chosen to approximately match the MACs from individual mobile net architectures. To support the partial execution of augmented branched mobile Net, we modified the TFLite interpreter. The exit path selection algorithm on the branched Mobilenet produced similar results as shown in the previous table. But the obvious benefit of this approach is that the application need not maintain multiple models. Moreover, multiple models need not be loaded to the RAM at the same time.

5.3 Results with image classification application

We noticed a significant decrease in the average number and the variation in the number of image buffers dropped during realtime image classification. This drop is at the cost of prediction accuracy. The result is shown in the table.

MobileNet version	Average frames dropped	Variation in frame dropped
mobilenet v2 1.4 224	36	15.2
mobilenet all (α = 0.5)	**28**	**11.8**

6 Implementation details

6.1 Mobile Workload automation

To carry out the experiments under consistent system background load, we automated the user interactions with the Android device. The interaction was recorded with the monkey recorder and ran with the monkey runner through ADB (1). A python scripts controlled the sequence in which these interactions are being made. Over multiple runs, we observe approximately the same CPU load trends, which verifies that the results can be reproducible.

6.2 Load tracking and load trend estimation

To record the active and idle CPU time, we read $/proc/stat$ file system at a fixed interval. The load tracking functionality was launched in a different background thread and was signaled by the main program for start and stop. To ensure that this thread does not cause the system to be loaded, we set the load tracking system to update its load every 100 milliseconds.

6.3 Static analysis of MACs through each branch

We created a tool to compute the number of MACs for each branch in the augmented mobile net architecture. The static analysis was run before feeding the model to the TFLite

interpreter. The MACs information is passed to the exit point selection algorithm to estimate the runtime for each exit points in the architecture.

6.4 Modification is TFLite interpreter

To enable the partial execution through TFLite interpreter when running augmented mobile net architecture, we modified the graph executer class. The invoke function was then modified to accept the tensor node-id argument. The graph executer ceases the computation once the supplied node id tensor is filled. An overview of the modified blocks is shown in the figure.

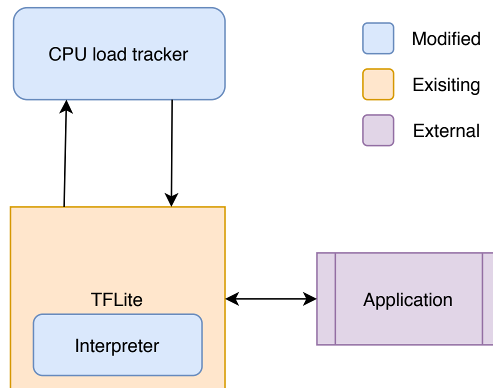

Figure 9: System overview diagram.

6.5 Frame drop count in image classification application

We modified the official image classification app (13) provided as an example in the TensorFlow repository and logged the number of frames dropped.

The source code used for this project can be found in our Github repository. To aid the reproducibility, we are updating the guide and the software dependencies.

7 Discussion and Conclusions

For reasons of security and efficiency much of machine learning inference has moved from cloud servers to on-edge, on-device computation. Many applications require fast, efficient predictions and have this need prioritized over highest accuracy. However, there are many computational challenges lurking around in this domain.

This paper discusses the two existing approaches for handling these dilemmas. The first seeks to minimize computation through techniques like architectural improvements, quantization. However, this approach fails to account for differences in single inference runtime caused by background load. The second approach involves computational branches using multiple exit points, like BranchyNet. In our approach outlined in this paper, we present a deterministic policy to select and terminate the computation through an early

exit point based on a specified QoS metric. This metric specifies if the desired model outcome favors accuracy or efficiency.

In this paper, we experimented with real-time image classification models from the mobile-net family of architectures. Six different models were tested, with varying CPU loads and QoS alpha values. Results of this experiment indicated that regardless of the model, average CPU load is a reliable indicator in predicting inference run-time. The next experiment involved modifying the mobilenet model to have multiple exit points. We compared the performance of the existing mobilenet model with our modified exit point version. Results of this experiment indicated that the modified version had a decrease in average number and variation of image buffers dropped during image classification.

References

Android. monkeyrunner. `https://developer.android.com/studio/test/monkeyrunner`, 2020. [Online; accessed].

Apple. coreml. `https://developer.apple.com/documentation/coreml`, 2020. [Online; accessed].

Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541, 2006.

Facebook. Caffe2. `https://research.fb.com/downloads/caffe2/`, 2020. [Online; accessed].

Google. tflite. `https://www.tensorflow.org/lite`, 2020. [Online; accessed].

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

Jesse Hostetler. Toward runtime-throttleable neural networks. *arXiv preprint arXiv:1905.13179*, 2019.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

S Obdržálek, Gregorij Kurillo, Jay Han, Ted Abresch, Ruzena Bajcsy, et al. Real-time human pose detection and tracking for tele-rehabilitation in virtual reality. *Studies in health technology and informatics*, 173:320–324, 2012.

Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 475–480. IEEE, 2016.

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast in-

ference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

Tensorflow. classification. https://www.tensorflow.org/lite/models/image_classification/overview, 2020. [Online; accessed].