Cloud-based or On-device: An Empirical Study of Mobile Deep Inference

Tian Guo
Computer Science Department
Worcester, MA, USA
Worcester Polytechnic Institute
tian@cs.wpi.edu

***Abstract*—Modern mobile applications are benefiting significantly from the advancement in deep learning, e.g., implementing real-time image recognition and conversational system. Given a trained deep learning model, applications usually need to perform a series of matrix operations based on the input data, in order to infer possible output values. Because of computational complexity and size constraints, these trained models are often hosted in the cloud. To utilize these cloud-based models, mobile apps will have to send input data over the network. While cloud-based deep learning can provide reasonable response time for mobile apps, it restricts the use case scenarios, e.g. mobile apps need to have network access. With mobile specific deep learning optimizations, it is now possible to employ on-device inference. However, because mobile hardware, such as GPU and memory size, can be very limited when compared to its desktop counterpart, it is important to understand the feasibility of this new on-device deep learning inference architecture. In this paper, we empirically evaluate the inference performance of three Convolutional Neural Networks (CNNs) using a benchmark Android application we developed. Our measurement and analysis suggest that on-device inference can cost up to two orders of magnitude greater response time and energy when compared to cloud-based inference, and that loading model and computing probability are two performance bottlenecks for on-device deep inferences.**

***Index Terms*—Mobile Deep Learning, Performance Measurement**

I. Introduction

Deep learning has started to gain popularity in powering up modern mobile applications. Training deep neural networks requires access to large amounts of data and computing powers. As a result, these neural networks are often trained by leveraging cheaper, yet more powerful cloud GPU clusters. Once trained, the inference phase can be completed in a reasonable amount of time, e.g., less than one second, using a single machine. Pre-trained models can be hosted for private use or offered as public cloud deep learning services [1], [2]. To utilize cloud-based pre-trained models, mobile app developers use exposed cloud APIs to offload deep learning inference tasks, such as object recognition shown in Figure 1, to the hosting server. Mobile apps that execute inference tasks this way is referred to as *cloud-based* deep inference.

Despite their increasing popularity, the use case scenarios of cloud-based deep inference can be limited due to data privacy concern, unreliable network condition, and impact on battery life. Alternatively, we can perform inference tasks locally using mobile CPU and GPU [3]. We refer to this mobile deep learning approach as *on-device* deep inference. On-device deep inference can be a very attractive alternative to the cloud-based approach, e.g., by providing mobile applications the ability to function even without network access.

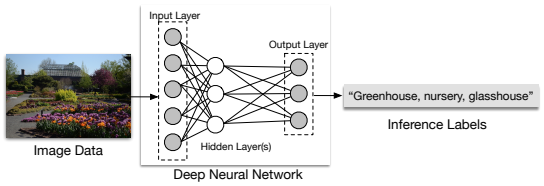


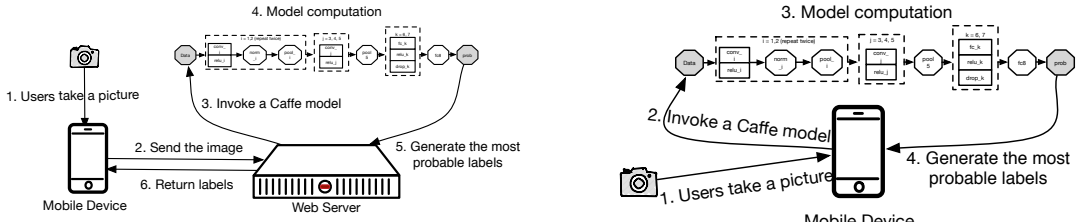
Fig. 1: Object recognition with deep neural networks. An image passes through a deep neural network that consists of layers of neurons. The output layer produces the inference labels that best describe the image.

Given the above two design choices for implementing deep inference, it is beneficial for developers to understand the performance differences. However, it is not straightforward to reason about mobile apps performance when using *on-device* deep inference. The difficulties can be attributed to the following reasons. First, deep neural networks (DNN) can differ vastly in terms of network architecture, number of parameters, and model sizes (see Figure 4). Second, the inference tasks can be of different complexities depending on the input data, e.g., large images vs. small images, and the DNN model in use. Third, mobile devices often have heterogenous resource capacities and can exhibit different runtime behaviors, such as garbage collection activities, given different deep learning models and inference tasks combinations.

To address the challenges of understanding deep learning inference, in this paper we develop a mobile application benchmark that allows end users to supply configurations including inference mode, model, and input data. We conduct a detailed measurement study using our mobile application with cloud-based and on-device deep inference, three convolutional neural networks, and a dataset of fifteen images. Our evaluation shows that cloud-based approach can save up to two orders of magnitude in terms of both end-to-end response time and mobile energy consumption. Further, we analyze the performance differences between on-device and cloud-based approaches with an in-depth analysis of performance bottlenecks. Our analysis suggests that loading and computing using deep learning models dominate the end-to-end inference time. Lastly, we find that current on-device approach, when utilizing mobile GPU, provides reasonable average inference time of 2.2 seconds if models are preloaded into the memory.

II. Background

In this section, we first provide a background on deep learning models and platforms. We then discuss two design choices for implementing deep learning powered mobile apps. Lastly,



(a) Cloud-based inference. **(b)** On-device inference.

Fig. 2: Design choices of deep learning powered mobile applications. We use our implemented object recognition Android App as an example to illustrate the steps involved to perform cloud-based inference and on-device inference.

we provide a brief explanation of mobile OS memory resource management and its associated performance implication.

A. Deep Learning Models

Deep learning refers to a class of artificial neural networks (ANNs) for learning the right representation of input data and is widely used for visual and speech recognition [4]. In this paper, we focus on a specific visual recognition task called object recognition that maps an image to a list of most probable text labels using deep neural networks (DNNs), as illustrated in Figure 1. DNNs usually consist of one input layer and one output layer, and a number of hidden layers. Each layer can consist of different numbers of processing units (neurons). Given an image, DNNs use it as input to initialize the first layer, pass it through processing units in hidden layers, and eventually generate a probability distribution over label categories at the final layer. The inference labels correspond to the categories with the highest probability values. The complexity of the inference tasks depends on the computation defined in each unit as well as the total number of units and layers.

In this paper, we focus on a special class of deep learning models called convolutional neural networks (CNNs) [5], [6], [7]. CNNs are widely used for visual recognition tasks due to their high accuracy. CNNs usually consist of many different types of layers, such as convolutional layers, pooling layers, and fully-connected layers (refer to Figure 4). Each layer takes data from previous layer and apply predefined computation in parallel. For example, convolutional layers use both linear convolutional filters and nonlinear activation functions to generate feature maps [8], while pooling layers control overfitting by reducing parameters in the representation [9].

There are a number of popular deep learning frameworks, such as Caffe [10], Torch [11], TensorFlow [12], that ease the training and deploying of deep learning models. Different frameworks require different syntaxes to describe CNNs and have different trade-offs for training and inference [13] phases. Because Caffe provides a flexible way to define CNN layers and has a large collection of pre-trained models [14], we choose to evaluate CNN models trained using the Caffe framework in this paper. A pre-trained Caffe model contains a binary `caffemodel` file that describes model parameters, a `prototxt` file that describes model network, and an accompanying text file of labels.

B. Mobile Apps

1) Cloud-based vs. on-device Inference: Deep learning powered mobile apps can be roughly categorized into two types (as shown in Figure 2) : the cloud-based inference and on-device inference. The key differences between these two architectures are where the CNN models are stored, and how the inference task is executed. Cloud-based inference leverages pre-trained models in powerful cloud servers and is by far the more popular design choice when it comes to designing mobile deep learning apps.

Alternatively, mobile apps can be built upon on-device inference. In essence, this means CNN models will be stored on mobile device and inference tasks will be executed using mobile CPU or GPU. Although conceptually simple, it is not always straightforward to deploy CNN models into mobile devices. For example, existing models that are designed to run on powerful servers from the outset can contain hundreds of layers and millions of parameters, therefore are not suitable to run on resource-constrained mobile devices. [1]

In this paper, we look at two existing approaches that enable on-device inference. The first approach relies on porting existing frameworks [18], [19] to mobile platforms so that CNN models can run on the mobile platform. When developing Android apps using Caffe Android Lib [19], app developers first need to compile different versions of `libcaffe.so` and `libcaffe_jni.so` for all supported mobile CPU and instruction sets. These compiled library files then need to be loaded before applications perform inference tasks. However, because mobile GPUs are very different from desktop GPUs, currently none of the ported mobile libraries support executing model computation using mobile GPUs. The second approach relies on third-party libraries that convert existing models to supported formats [3] to take advantage of the mobile GPU. For example, CNNDroid expresses CNN layers in `RenderScript` kernels so that the RenderScript runtime framework can parallelize model computations across both CPUs and GPUs [20].

2) Lifecycle Management and Its Performance Implication: When an user first launches the mobile app, Android OS will first call the `onCreate()` method inside the launcher main activity. After successfully setting up and initializing states, the activity runs in the foreground of the screen. This running activity is called foreground activity and at any given time, there is only one such activity. Android Runtime (ART) automatically manages application memory by using concurrent mark sweep (CMS) garbage collection algorithm that is optimized for interactive applications. By default, Android

[1]Mobile-specific platforms [15], [16] and mobile-specific models [17], designed by companies such as Google and Facebook, can be a promising approach towards efficient on-device inference for new mobile hardwares.

TABLE I: Hardware Specification of mobile device and cloud server used in our evaluation.

Inference Engine	CPU	GPU	Mem.	Storage	Battery
Nexus 5	2.26 GHz quad-core (Krait 400)	Adreno 330 (129.8 GFLOPS)	2GB	16GB	2300 mAh /8.74Wh
g2.2xlarge	2.6 GHz eight-core (Intel Xeon E5-2670)	NVIDIA GRID K520 (2448.4 GFLOPS)	15GB	60GB	N/A

TABLE II: Summary of image data sets. Each row describes the image dimension in terms of width by height, and the image size in KB.

Dataset	Dimension(WxH)/ Size(KB)					
1	2160 x 3840	785	1362	1396	1528	2599
2	1080 x 1920	251	479	547	739	576
3	540 x 960	82	146	165	196	226

OS allocates a heap size indicated by `getMemoryClass()` method. For memory-intensive apps, such as CNN based mobile apps, we can request to run the app with large heap. By doing so, Android OS might allocate a heap size indicated by `getLargeMemoryClass()` method. For example, in Nexus 5, we can increase the application heap from the default 192 MB to 512 MB with large heap option turned on. However, for memory constrained mobile devices, Android OS might still allocate the default heap size to applications. When an activity is in the background, it can be killed by Android OS when memory is needed elsewhere, e.g. by another running activity [21]. Since a CNN model can be up to hundreds of MBs, and therefore apps of on-device inference architecture are more likely to be killed by OS in a memory-constrained mobile device to free up memory. When users need to interact with these killed apps, the app will have to be completely restarted and restored to its previous state, incurring undesirable startup latency.

III. Mobile Benchmark implementation

We implement an object recognition Android app that supports both cloud-based and on-device inference. As shown in Figure 2, our Android app takes images as data input, invokes one of the CNN models, generates a probability distribution over labels, and displays the top five most probable labels to the user.

In the cloud-based inference mode, image data needs to be sent from our mobile app to an Apache web server hosted inside the Amazon Virginia data center. To achieve this, our mobile app specifies both the IP address of the cloud server and the `php` script name when creating the HTTP connection. After the HTTP connection is successfully established, our mobile app will send the original or downscaled image of 224 by 224 pixels in dimension, to the web server. This downsizing step is because all three deep learning models we are using only require bitmaps of the scaled dimension. After the image finishes uploading, the web server will then invoke the specified Caffe model, use either CPU-only or GPU accelerated Caffe framework for probability computation, and return the top five labels to the mobile app.

In the on-device inference mode, both the selected CNN model and the image bitmap object are loaded into mobile device's memory. Depending on which framework is selected, our mobile app will use either CPU-only Caffe Library or GPU-enabled CNNDroid to generate the probability distribution over labels.

IV. Mobile Deep Inference Evaluation

A. Experimental Setup

Our evaluation consists of empirical measurement of performing inference tasks using both cloud-based and on-device deep learning models, as illustrated in Figure 2. We use our implemented mobile deep inference benchmark application, described in Section III, to perform both inference modes. For evaluating cloud-based setup, we first install Apache web server on a cloud server, and then store three CNN models in the web server. We select the cheapest GPU server, i.e., `g2.2xlarge`, from Amazon Virginia data center for both the CPU and GPU cloud-based inference. All our cloud servers run Ubuntu 14.04. For evaluating on-device inference, we use a LG Nexus 5 (late 2013) mobile phone that runs Android 6.0.1 Marshmallow and is on university campus Wi-Fi. Hardware specifications can be found in Table I.

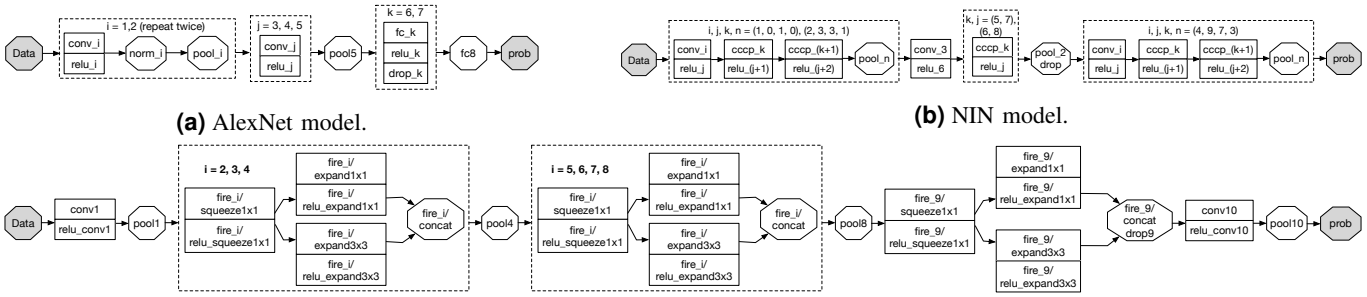
We selected three Caffe-based CNN models: AlexNet [5], NIN [6], and SqueezeNet [7], as shown in Figure 4. We choose to evaluate our mobile application using these models because they provide very similar top-5 error rates[2] on ImageNet data set, while differ vastly in terms of model sizes. All three models only require images of dimension 224 by 224 pixels. Our inference input data set contains three groups of a total fifteen images, as summarized in Table II. Each group consists of images of the same dimension. The second and third groups are generated by resizing the original images in the first group by scale factors of two and four respectively.

To measure the time taken to perform each step in an inference task, we instrumented our Android app to output event timestamps to a log file. We use `Logcat`, a command-line tool, to pull log files of each experiment run from the Nexus 5 through a laptop running within the same university campus network over Wi-Fi. To prevent inaccurate power measurement, we avoided connecting the Nexus 5 to laptop through USB in our experiments. We also ensure the mobile device is fully charged at 100% at the start of every measurement session. `Logcat` logs contains system events, including Android Runtime (ART) garbage collection logs, and application-level logs.

To measure the power consumption and resource utilization of our mobile app, we use the `Trepn` profiler [24] and save the profile results in `csv` format for offline analysis. We follow the best practices to reduce a profiler's impact on measurement inaccuracy by configuring `Trepn` to sample every 100 ms for three data points of interests: battery consumption, normalized CPU usage, and the GPU load.

For each experiment run, the user first launches the `Trepn` profiler mobile app. Inside the profiler app, the user can select to profile our developed object recognition app. This will automatically launch our app, and present the UI for setting up experiment configuration. The user can select either to run the experiment in cloud or mobile modes, indicate the deep learning models to use, and choose the dataset to perform the object recognition on.

[2]Top-5 error rate is defined as the percentage of incorrectly labeled test images in the top five most probable labels.



(a) AlexNet model.

(b) NIN model.

(c) SqueezeNet model.

Fig. 3: Neural Network architecture visualization. For each neural network, we show a simplified version based on a web tool [22]. These three convolution neural networks achieve very close top-5 ImageNet accuracy of around 80% [7], [23], but have vastly different model sizes. AlexNet [5] is 233 MB, NIN [6] is 29 MB, and SqueezeNet [7] is 4.8 MB.

TABLE III: Comparisons of cloud-based and on-device deep inference. Cloud-based models outperform on-device models by at least six times. The model loading time dominates the object recognition task, and the time taken to compute the label probabilities dominates the inference time for on-device mode. In addition, on-device mode consumes twice as much battery, therefore consumes at least 12 times as much mobile energy.

Inference Mode	Object Recognition Time Breakdown [ms]				On-device Resource Consumption			
	Load model	Rescale bitmap	Upload bitmap	Compute probability	CPU [%]	GPU [%]	battery [mW]	Mem[MB]
Cloud+CPU	0.00	76.16	36.83	238.60	6.22	0.89	1561.63	1279.10
Cloud+GPU	0.00	76.16	36.83	18.60	6.36	0.43	1560.17	1311.38
Device+Caffe	2422.13	79.98	0.00	8910.64	35.01	0.14	3249.01	1637.16
Device+CNNDroid	61256.17	70.43	0.00	2131.70	22.20	27.14	2962.58	1752.45

B. End-to-end Comparisons

In this section, we present the measurement results of object recognition time and mobile resource utilization for both cloud-based and on-device inference.

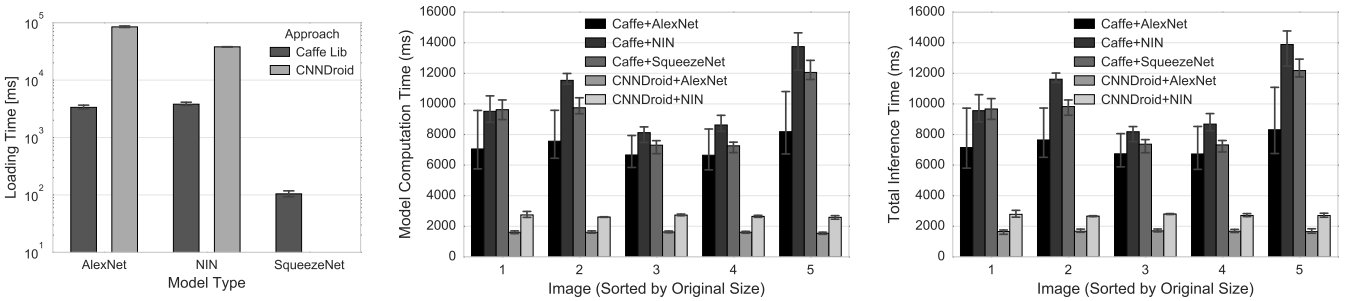
1) Object Recognition Time: Table III summarizes the average end-to-end performance and resource consumption of executing object recognition using both cloud-based and on-device inference modes. The task of object recognition is further broken down into four steps: loading CNN models into memory, downscaling image input to desired dimension, uploading input data to the cloud server, and computing the probability matrix. For each inference mode, we repeat the recognition tasks using all fifteen images and three CNN models. We measure the time to execute each step and calculate the average. We use CPU-only Caffe framework and GPU accelerated Caffe framework for toggling the CPU and GPU mode in our `g2.2xlarge` server. Note, the time to load models is negligible in the cloud-based scenario because models already reside inside the memory and can be used to execute the inference task immediately. Similarly, on-device mode does not incur any time for uploading image bitmaps. Recall, the inference time is the sum of rescaling, uploading bitmap and computing probability over one bitmap, and the recognition time is the sum of amortized model loading time over a batch of images and inference time. The average cloud-based inference time is 351.59 ms/131.59 ms when using CPU-only/GPU of a well-provisioned cloud instance hosted in a nearby data center. As shown, because inference tasks are typically data-parallel and therefore can be accelerated by up to 10x when using GPU. However, we should note that such results represent a lower bound performance of real-world setting. In a real-world deployment scenario, object recognition time can last much longer due to reasons such as overloaded cloud servers and variable mobile network conditions. The total inference time when running on-device is almost 9 seconds when using Caffe-based model, and 2.2 seconds when using CNNDroid model.

2) Energy and Resource Consumption: Table III shows the resource consumption of mobile device when running the object recognition mobile application. As a baseline, we measure the performance when the device is idle and the screen is turned on. The CPU utilization and power consumption is 3.17% and 1081.24 mW respectively. Cloud-based mode consumes roughly the same amount of CPU and 44.4% more power consumption comparing to the baseline. However, on-device mode not only incurs significantly higher CPU utilization (and in the case of CNNDroid, GPU utilization as well), but also require two times more power consumption when compared to the baseline. In all, we can calculate the energy consumption of different inference modes by multiplying the average inference (recognition) time by the average power consumption. Cloud-based inference requires as low as 0.057 mWh energy when using faster GPU computation, and on-device based inference consumes up to 8.11 mWh.

Result: *Cloud-based inference exhibits substantial benefits in terms of inference response time and mobile energy savings over on-device inference, in this case by two orders of magnitude. This is due to more powerful processing power and shorter durations of inference.*

C. On-device Deep Inference Performance Analysis

In this section, we analyze the performance differences by dissecting the on-device object recognition task with an in-depth study of time breakdown, and resource utilization. We focus on understanding the performance of on-device



(a) Model loading time. **(b)** Model computation time. **(c)** Total inference time.

Fig. 4: On-device Inference Time. We compare two approaches for device-based inference using three CNN models. When using CNNDroid-based approach, trained models need to be converted to supported format.

TABLE IV: Summary of GC activities when using CNNDroid-based device inference. ART uses the default CMS GC, and the GC time takes up to 9.89% during model loading, and up to 25% during user interactions. The average GC pause time can be up to 39.23 ms.

On-device Inference	Phase	Duration[ms]	Num. of GC	GC Time [ms]	GC Pause [ms]
CNNDroid+AlexNet	Load	84537.33	4.33	513.55	10.42
CNNDroid+NIN	Model	37975	16.67	3757.30	175.76
CNNDroid+AlexNet	User	11800	4	536.55	4.60
CNNDroid+NIN	Interaction	17166.67	7	4307.18	274.66

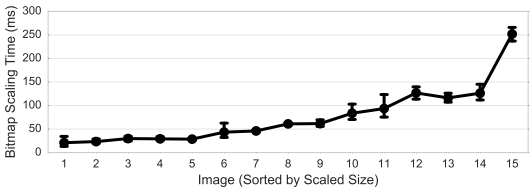


Fig. 5: Bitmap downscaling time. The time taken to downscale the image in the mobile device grow with the image size. Specially, larger images also experiences proportionally longer scaling time because of the limited memory resources assigned to the mobile application.

deep inference and the implications for potential peformance improvement.

1) Impact of Deep Learning Models: The choice of deep learning models including the framework and CNN model design can have a significant impact on the performance, due to different model sizes and complexities. We quantify such impacts with both the model loading and probability computation time. We plot the loading time in Figure 4(a) in *log* scale. For loading the same model (AlexNet and NIN), the ported Caffe library takes up to 4.12 seconds, about 22X faster than using CNNDroid. Furthermore, it only takes an average of 103.7 ms to load the smallest SqueezeNet model.[3] This loading happens whenever users first launch the mobile application, and potentially when a suspended background app is brought back. Our measurement of CNNDroid's long loading time suggests that users need to wait for up to 88 seconds to be able to interact with the mobile app. Although long loading time can be amortized by the number of inference tasks during one user interaction session, it still negatively impact user experiences.

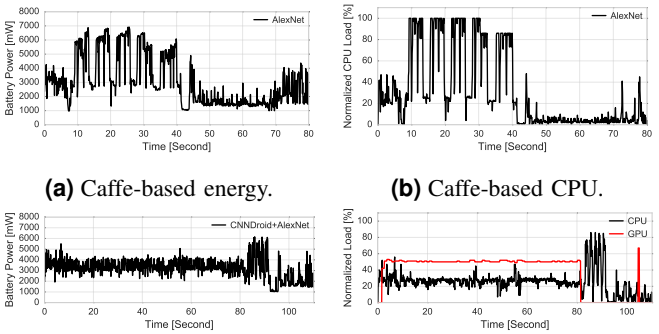
Next, we show the time taken to compute the input image using five different configurations in Figure 4(b). For each configuration, we measure the computation time taken for all five images and collect a total of 75 data points. Each bar represents the average computation time across three versions of the same image and the standard deviation. CNNDroid-based AlexNet inference achieves the lowest average of 1541.67 ms, compared to the longest time of 13745.33 ms using ported Caffe NIN model. Even with the fastest device-based inference, it still takes three times more than CPU-based cloud inference. In addition, we plot the end-to-end inference time in Figure 4(c). This total inference time includes the bitmap scaling time, the GC time, and the model computation time. CNNDroid-based approach takes an average of 1648.67 ms for performing object recognition on a single image, about seven times faster than using ported Caffe models. Based on the response time rules [25], [26], it might lead to poor user experiences when using certain device-based inference approach.

2) Impact of Limited Mobile Memory: During loading CNNDroid-based models, we observe much more frequent, and long lasting garbage collecting activities performed by Android Runtime in our mobile device. When running our app using CNNDroid library, we have to request for a large heap of 512 MB memory.[4] Even with a large heap, the memory pressure of creating new objects has lead to a total of 8.33 (23.67) GC invocations when using CNNDroid-based AlexNet (NIN) model, as indicated in Table IV. Our evaluation suggests that by allocating more memory to deep learning powered mobile apps, or running such apps in more powerful mobile devices can mitigate the impact of garbage collection.

3) Impact of Image Size: Because the CNN models only require images of dimension 224 by 224 pixels to perform inference tasks, we can scale input image to the required dimension before sending. Figure 5 shows the time taken to scale images with different sizes. Each data point represents the average scaling time across five different runs. The time taken to resize image grows as its size increases. It is only beneficial to downscale an image of size x_1 to x_2 if $T_d(x_1, x_2) + T_n(x_2) \leq T_n(x_1)$, where $T_d(x, y)$ represents

[3]We did not have results for running SqueezeNet using CNNDroid library because CNNDroid library currently does not support newer deep neural networks that include expand convolution layers, such as SqueezeNet.

[4]Running the app with the default 192 MB memory will lead to `OutOfMemoryError`.



(a) Caffe-based energy. **(b)** Caffe-based CPU.

(c) CNNDroid-based energy. **(d)** CNNDroid-based resource.

Fig. 6: Energy consumption and resource utilization of on-device object recognition task.

the time to downscale an image from size x to y and $T_n(x)$ denotes the time to upload an image of size x to a cloud server. For example, based on our measurement, it takes an average of 36.83 ms to upload an image of 172 KB to our cloud server. Also, from Figure 5, we know that it takes up to 38 ms to resize an image less than 226 KB. By combining these two observations, it is easy to conclude that directly uploading image one to five is more time efficient. We can expect to make informed decisions about whether resizing an image of size x before uploading is beneficial or not given enough time measurements of resizing and uploading steps.

Result: *Our analysis shows that on-device inference's performance bottlenecks mainly exhibit in loading model and computing probability steps.*

D. On-device Deep Inference Resource and Energy Analysis

In Figure 6, we analyze both the energy consumption and resource utilization when running our app in different configurations . We compare the time-series plots of running AlexNet model using Caffe Android library and CNNDroid framework. The plots correspond to experiment runs that perform inference tasks on image set one.

For Caffe Android library based approach, we observe an initial energy consumption (and CPU utilization) that increases corresponding to loading AlexNet CNN model into the memory, a continuation of energy spike during model computation, and the last phase that corresponds to displaying images and the most probable label texts, in Figure 6(a) and Figure 6(b). The other two CNN models, NIN and SqueezeNet, exhibit very similar usage pattern[5]. Specifically, in the case of NIN, the initial model loading causes the energy consumption to increase from baseline 1081.24 mW to up to 5000 mW; when performing the model computation, both the energy consumption and CPU utilization spikes to more than 7000 mW and 66.2%. Note in the case of SqueezeNet, we only observe a very small window of both energy and CPU spikes at the very beginning of measurement. This is because SqueezeNet can be loaded in 109 ms, compared to more than 3000 ms to load either AlexNet or NIN.

In contrast, we observe two key usage differences in CNNDroid approach, as shown in Figure 6(c) and Figure 6(d). First, CNNDroid-based AlexNet exhibits a longer period of more stable and lower energy consumption compared to its counterpart in Caffe-based approach. This is mainly because CNNDroid explicitly expresses some of the data-parallel workload using `RenderScript` and is able to offload these workload to more energy-efficient mobile GPU [28] (indicated by the high GPU utilization during model loading). Second, the total model computation time is significantly shortened from 40 seconds to around five seconds. In all, by shifting some of computation tasks during model loading, CNNDroid-based approach successfully reduces the user perceived response time. However, the CNNDroid approach consumes 85.2 mWh energy, over 42% more than Caffe-based approach. Note 91% of CNNDroid energy is consumed during model loading phase, and therefore can be amortized by performing inference tasks in batch.

Result: *The CNNDroid-based approach is more energy-efficient in performing inference tasks compared to the Caffe-based approach when models are preloaded into the mobile memory.*

[5]Interested readers can refer to our arXiv version [27] for additional results of Caffe-based NIN and SqueezeNet models, and CNNDroid-based NIN.

V. Related Work

To better understand the performance and power characteristics of modern mobile applications, researchers have developed a number of performance monitor tools, such as 3GTest [29], 4GTest [30], AppInsight [31], eprof [32], and Trepn [24] over the past decade. Our paper focuses on understanding the performance bottlenecks of a new class of mobile applications that are powered by deep learning models, with an implemented object recognition benchmark Android app.

In order to improve response time and preserve batter energy in resource-constrained devices, researchers have proposed to offload computation intensive tasks to the cloud [33], [34], [35], [36]. Our paper confirms that cloud-based inference mobile apps still deliver better response time and consume less power compared to mobile-based counterparts. However, recent development of mobile specific deep learning optimizations [37], [38], [3], [17], [15], [16] and improvement in mobile GPU energy consumption [28] are promising improvements towards efficient mobile-based deep learning apps. Our work can be easily extended to evaluate the efficiency of these new models and hardware.

VI. Conclusion

In this paper, we evaluate the current approaches to perform deep inference tasks in resource-constrained mobile devices. Our analysis show that while cloud-based inference incurs reasonable response time and energy consumption, current on-device inference is only feasible for very limited scenarios. However, with both industry and research efforts on adapting deep neural networks to the mobile devices, we believe it is very likely that on-device inference can be done efficiently in the near future. Consequently, when developing deep learning powered mobile apps, developers will have the freedom to choose from cloud-based, device-based or even a hybrid approach.

Acknowledgements. We thank all the reviewers for their insightful comments and Sam Ogden for proofreading, which improved the quality of this paper.

REFERENCES

[1] G. C. Platform, "Cloud vision api," Accessed on 2017.

[2] clarifai, "IMAGE AND VIDEO RECOGNITION API," Accessed on 2017.

[3] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "Cndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android," in *Proceedings of the 2016 ACM on Multimedia Conference*, ser. MM '16, 2016, pp. 1201–1205.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[6] S. Y. M. Lin, Q. Chen, "Network in network," *International Conference on Learning Representations, 2014 (arXiv:1312.4400v3)*, 2014.

[7] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *arXiv:1602.07360*, 2016.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Intelligent Signal Processing*. IEEE Press, 2001, pp. 306–351.

[9] S. CS231n, "Convolutional neural networks for visual recognition." http://cs231n.github.io/convolutional-networks/, 2017.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[11] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[13] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv:1511.06435*, 2015.

[14] B. Caffe, "Model zoo," https://github.com/BVLC/caffe/wiki/Model-Zoo, Accessed on 2017.

[15] TensorFlow, "Introduction to tensorflow lite," https://www.tensorflow.org/mobile/tflite/, Accessed on 2018.

[16] Y. Jia and P. Vajda, "Delivering real-time ai in the palm of your hand," https://code.facebook.com/posts/196146247499076/delivering-real-time-ai-in-the-palm-of-your-hand/, Nov. 8, 2016.

[17] M. Z. Andrew G. Howard, "Mobilenets: Open-source models for efficient on-device vision," https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html, June 14, 2017.

[18] soumith, "Torch-7 for android," https://github.com/soumith/torch-android, Accessed on 2017.

[19] sh1r0, "Caffe android lib," https://github.com/sh1r0/caffe-android-lib, Accessed on 2017.

[20] G. A. Developer, "Renderscript api guides," https://developer.android.com/guide/topics/renderscript/compute.html, Accessed on 2017.

[21] Google Android Developer, "Activity," https://developer.android.com/reference/android/app/Activity.htm, Accessed on 2017.

[22] NetScope, "A web-based tool for visualizing neural network architectures," http://ethereon.github.io/netscope/quickstart.html, Accessed on 2017.

[23] BVLC Caffe, "Models accuracy on imagenet 2012 val," https://github.com/BVLC/caffe/wiki/Models-accuracy-on-ImageNet-2012-val, 2015.

[24] Q. T. Inc., "Trepn power profiler," https://developer.qualcomm.com/software/trepn-power-profiler, Accessed on 2017.

[25] J. NielSen., "Website response times." https://www.nngroup.com/articles/website-response-times/, June 21, 2010.

[26] appdynamics, "16 metrics to ensure mobile app success," Accessed on 2017.

[27] T. Guo, "Towards efficient deep inference for mobile applications," *arXiv:1707.04610*, 2017.

[28] nvidia, "Gpus are driving energy efficiency across the computing industry, from phones to super computers." http://www.nvidia.com/object/gcr-energy-efficiency.html, Accessed on 2017.

[29] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814452

[30] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 225–238. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307658

[31] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild."

[32] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 29–42.

[33] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[34] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[35] P. Angin, B. Bhargava, and R. Ranchal, "Tamper-resistant autonomous agents-based mobile-cloud computing," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 843–847.

[36] T. Guo, P. Shenoy, K. K. Ramakrishnan, and V. Gopalakrishnan, "Latency-aware virtual desktops optimization in distributed clouds," *Multimedia Systems*, Mar 2017. [Online]. Available: https://doi.org/10.1007/s00530-017-0536-y

[37] N. D. Lane, S. Bhattacharya, and P. Georgiev, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2016, pp. 1–12.

[38] L. N. Huynh, R. K. Balan, and Y. Lee, "Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices," in *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, ser. WearSys '16. New York, NY, USA: ACM, 2016, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2935643.2935650