MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints

Seungyeop Han[*]
University of Washington

Haichen Shen[*]
University of Washington

Matthai Philipose
Microsoft Research

Sharad Agarwal
Microsoft Research

Alec Wolman
Microsoft Research

Arvind Krishnamurthy
University of Washington

ABSTRACT

We consider applying computer vision to video on cloud-backed mobile devices using Deep Neural Networks (DNNs). The computational demands of DNNs are high enough that, without careful resource management, such applications strain device battery, wireless data, and cloud cost budgets. We pose the corresponding resource management problem, which we call *Approximate Model Scheduling*, as one of serving a stream of heterogeneous (i.e., solving multiple classification problems) requests under resource constraints. We present the design and implementation of an optimizing compiler and runtime scheduler to address this problem. Going beyond traditional resource allocators, we allow each request to be served *approximately*, by systematically trading off DNN classification accuracy for resource use, and *remotely*, by reasoning about on-device/cloud execution trade-offs. To inform the resource allocator, we characterize how several common DNNs, when subjected to state-of-the art optimizations, trade off accuracy for resource use such as memory, computation, and energy. The heterogeneous streaming setting is a novel one for DNN execution, and we introduce two new and powerful DNN optimizations that exploit it. Using the challenging continuous mobile vision domain as a case study, we show that our techniques yield significant reductions in resource usage and perform effectively over a broad range of operating conditions.

1. INTRODUCTION

Over the past three years, Deep Neural Networks (DNNs) have become the dominant approach to solving a variety of computing problems such as speech recognition, machine translation, handwriting recognition, and computer vision problems such as face, object and scene recognition. Although they are renowned for their excellent recognition performance, DNNs are also known to be computationally intensive: networks commonly used for speech, visual and language understanding tasks routinely consume hundreds of MB of memory and GFLOPS of computing power [30, 46], typically the province of servers (Table 1). However, given the relevance of such applications to the mobile setting, there is a strong case for executing DNNs on mobile devices. In this paper, we present a framework for executing *multiple* applications that use large DNNs on (intermittently) cloud-connected mobile devices to process *streams* of data such as video and speech. We target high-end mobile-GPU-accelerated devices, but our techniques are useful broadly. We require little specialized knowledge of DNNs from the developer.

	face [46]	scene [52]	object [44]
training time (days)	3	6	14-28
memory (floats)	103M	76M	138M
compute (FLOPs)	1.00G	2.54G	30.9G
accuracy (%)	97	51	94

Table 1: Although DNNs deliver state-of-the art classification accuracy on many recognition tasks, this functionality comes at very high memory (space shown is to store an active model) and computational cost (to execute a model on a *single* image window).

Recent work has attacked the overhead of DNNs from several directions. For high-value scenarios such as speech and object recognition, researchers have crafted efficient DNNs by hand [32, 42, 45] or structured them for execution on efficient co-processors such as DSPs (for smaller DNNs) [31]. In the somewhat longer term, work from the hardware community on custom accelerators seems very promising [11]. An intermediate approach that is both automatic and effective is what we term *model optimization*, which are techniques that apply automatically to any DNN and reduce memory [13, 22, 50, 51] and processing demands [25, 42] of DNNs, typically at the cost of some classification accuracy. These efforts have yielded promising results: it is possible to generate models that sacrifice a modest amount of classification accuracy (e.g., 1-3%) that are small enough to fit comfortably in mobile memories (e.g., 10× reduction in space used) and can be executed in real time (e.g., 3× reduction processing demands) on a mobile GPU.

MCDNN considers applying model optimization to the multi-programmed, streaming setting. We anticipate that in the near future, multiple applications will seek to run multiple DNNs on incoming high-datarate sensor streams such as video, audio, depth and thermal video. The large number of simultaneous DNNs in operation and the high frequency of their use will strain available resources, even when optimized models are used. We use two insights to address this problem. First, model optimization typically allows a graceful tradeoff between accuracy and resource use. Thus, a system could adapt to high workloads by using less accurate variants of optimized models. Second, both streaming and multi-programming themselves provide additional structure that enable powerful new model optimizations. For instance, streams often have strong temporal locality (e.g., a few classes dominate output for stretches of time), and queries from

[*]Both authors contributed equally to this work.

MobiSys'16, June 25-30, 2016, Singapore, Singapore
© 2016 ACM. ISBN 978-1-4503-4269-8/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2906388.2906396

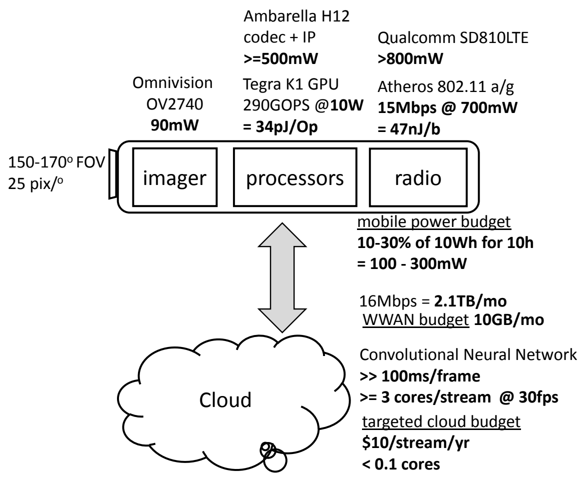


Figure 1: Basic components of a continuous mobile vision system.

multiple programs on the same stream may have semantic similarity that allows sharing of computation.

We formulate the problem of adaptively selecting model variants of differing accuracy in order to remain within per-request resource constraints (e.g., memory) and long-term constraints (e.g., energy) while maximizing average classification accuracy as a constrained optimization problem we call *Approximate Model Scheduling (AMS)*. To solve AMS, MCDNN contains three innovations. First, we generate optimized variants of models by automatically applying a variety of model optimization techniques with different settings. We record accuracy, memory use, execution energy, and execution latency of each variant to form a *catalog* for each model. Second, we introduce two new model optimizations, specialization and sharing, that are designed to take advantage of streaming and multi-programming respectively. Finally, we provide a heuristic scheduling algorithm for solving AMS that allocates resources proportionally to their frequency of use and uses the catalog to select the most accurate corresponding model variant.

As a running case study, and for purposes of evaluation, we target the continuous mobile vision setting: in particular, we look at enabling a large suite of DNN-based face, scene and object processing algorithms based on applying DNNs to video streams from (potentially wearable) devices. We consider continuous vision one of the most challenging settings for mobile DNNs, and therefore regard it as an adequate evaluation target. We evaluate MCDNN on very large standard datasets available from the computer vision community. Our results show that MCDNN can make effective trade-offs between resource utilization and accuracy (e.g., transform models to use 4× fewer FLOPs and roughly 5× less memory at 1-4% loss in accuracy), share models across DNNs with significant savings (e.g., 1-4 orders of magnitude less memory use and 1.2-100× less compute/energy), effectively specialize models to various contexts (e.g., specialize models with 5-25× less compute, two orders of magnitude less storage, and still achieve accuracy gains), and schedule computations across both mobile and cloud devices for diverse operating conditions (e.g., disconnected operation, low resource availability, and varying number of applications). To the best of our knowledge, MCDNN is the first system to examine how to apply DNNs efficiently to streams.

2. CONTINUOUS MOBILE VISION

Continuous mobile vision (CMV) refers to the setting in which a user wears a device that includes a continuously-on (we target ten hours of continuous operation) camera that covers their field of view [6, 19, 37, 39, 47]. The device is often a custom wearable such as Google Glass, but possibly just a mobile phone in a pocket. Video footage from the camera is analyzed, typically in real time, using computer vision techniques to infer information relevant to the user. While current systems are research efforts aimed at key niche applications such as cognitive assistance for the elderly and navigational assistance for the blind, in the near future we anticipate a *multi-programming* setting aimed at the general consumer, where multiple applications issue distinct queries simultaneously on the incoming stream. For example, various applications may need to recognize what the wearer eats, who they interact with, what objects they are handling as part of a task, the affect of the people they interact with and the attributes of the place they are in. In this section, we introduce the resources involved in such a system and motivate careful resource management via controlling the overhead of Deep Neural Network (DNN) execution.

Video processing itself usually involves some combination of *detecting* regions of interest in each frame (e.g., faces, pedestrians or objects), *tracking* the trajectory of detected regions across time and *recognizing* the detailed identities and attributes of these regions (e.g., recognizing the identity of people and objects interacted with). Although traditionally detection and tracking have been performed with relatively lightweight algorithms, state-of-the-art variants of these are switching to Deep Neural Networks (DNNs) [29, 33]. More recent work has begun to integrate detection and recognition using DNNs [41]. Since detection and tracking computations are expected to be performed frequently (e.g., a few times a second), we expect DNNs to be applied several times to many of the video frames. We therefore view DNN execution as the bulk of modern vision computation.

Figure 1 sketches the architecture of a state-of-the-art mobile/cloud Continuous Mobile Vision (CMV) system. The two main physical components are a battery-powered mobile device (typically some combination of a phone and a wearable) and powered computing infrastructure (some combination of a cloudlet and the deep cloud). The camera on the wearable device must usually capture a large field of view at high resolution and moderate frame rate. A resolution of 4k (4096×2160 pixels) at 15 frames per second is not unreasonable[1], drawing 90mW from a modern imager.

Consider performing all vision in the cloud, a "pure off-loading" architecture. For high-resolution video, spending 0.5W on compression and 0.7-1W on wireless offload is conservative, yielding a total average power draw of 1.3 to 1.6W for imaging, compression and communication. A realistic 100× compression yields a 16Mbps stream (= 4096×2160×15×1.5×8, using the 1.5 byte-per-pixel YUV representation), or roughly 2.1TB per month at 10 hours usage per day. Finally, we assume a conservative 1 DNN application per frame (we expect that, in practice, applications may run multiple DNNs on incoming frames). Additionally assuming a conservative 100ms execution latency (we have measured 300-2000ms latencies for models commonly in use) for a DNN on a single CPU, keeping up with 15-30fps will require *at least* 1.5-3 cores, and often many times more.

In comparison, a large 3Ah mobile phone battery of today yields roughly 1.2W over 10 hours. Further, today's consumer mobile plans cap data use at 10GB per month. Finally, continuous use of the required cores will cost roughly $150-300 per year[2]; more realistic workloads could easily be 10 times as costly. Offloading *all* data for cloud-processing using maximally accurate DNNs would thus probably only be practical in usages with a large dedicated wearable battery, lightly subscribed WiFi connection (thus limiting mobility) and substantial cloud budget. One path to reducing these resource

[1]Better vertical coverage (e.g., 4096×4096 pixels total) would be preferable, perhaps from *two* standard high-resolution imagers.

[2]Assuming a 3-year upfront lease on a C4.large machine from Amazon EC2. GPU-based costs are at least as high.

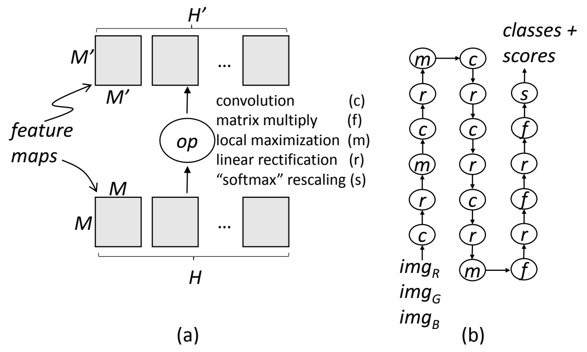


Figure 2: (a) DNN "layers" are array operations on lists of arrays called feature maps. (b) A state-of-the-art network for scene recognition, formed by connecting layers.

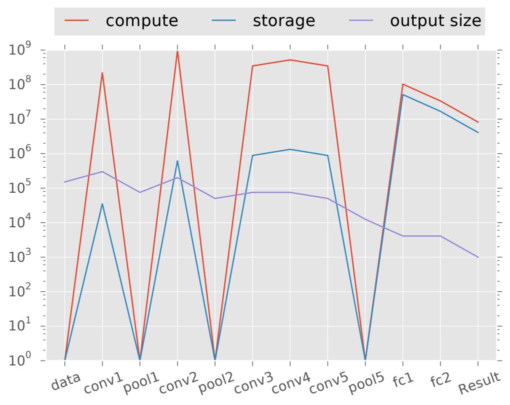


Figure 3: Resource usage of AlexNet across layers (note log scale).

demands is to reduce the amount of data transmitted by executing "more lightweight" DNN calculations (e.g. detection and tracking) on the device and only transmitting heavier calculations (e.g., recognition) to the cloud. Once at the cloud, reducing the overhead of (DNN) computations can support more modest cloud budgets.

Now consider performing vision computations on the device. Such local computation may be *necessary* during the inevitable disconnections from the cloud, or just *preferable* in order to reduce data transmission power and compute overhead. For simplicity, let us focus on the former ("purely local") case. Video encoding and communication overhead would now be replaced (at least partially) by computational overhead. Given mobile CPU execution speeds (several seconds of execution latency for standard DNNs), we focus on using mobile GPUs such as the NVIDIA Tegra K1 [3], which supports 300GFLOPS peak at a 10W *whole-system-wide* power draw. We time DNN execution on the K1 to take 100ms (for the "AlexNet" object recognition model) to 900ms (for "VGGNet"). Handling the aforementioned conservative processing rate of 1 DNN computation per frame at 15-30fps would require 1.5-30 mobile GPUs, which amounts to 15-300W of mobile power draw on the Jetson board for the K1 GPU. Even assuming a separate battery, a roughly 1-1.2W continuous power draw is at the high end of what is reasonable. Thus, it is important to substantially reduce the execution overhead of DNNs on mobile GPUs.

To summarize, significantly reducing the execution costs of DNNs can enable CMV in several ways. Allowing detection and tracking to run on the mobile devices while infrequently shipping to cloud can make transmission power and data rates manageable. Reducing the cost of execution in the cloud can make dollar costs more attractive. Allowing economical purely local execution maintains CMV service when back-end connectivity is unavailable. Finally, if execution costs are lowered far enough, pure local execution may become the default.

3. DEEP NEURAL NETWORKS

We now examine Deep Neural Networks, specifically the most commonly used variant called Convolutional Neural Networks (CNNs) in some detail. Our goal is to convey the basic structure of CNNs, their resource consumption patterns, the main current proposals for optimizing their resource consumption and further opportunities opened by streaming, multi-programmed settings.

3.1 Model structure

A CNN can be viewed as a dataflow graph where the nodes, or *layers*, are array-processing operations (see Figure 2(a)). Each layer takes as input a set of arrays (called *feature maps*), performs an array operation on them, and outputs a set of feature maps that will in turn be processed by downstream layers. The array operations belong to a small set, including matrix multiplication (that multiplies feature maps by a weight matrix), convolution (that convolves inputs by a convolution kernel, typically of size 3×3 or 5×5), max-pooling (that replaces each input array value by the maximum of its neighbors), non-linearizing (that replaces each array value by a non-linear function of itself) and re-scaling (that re-scales inputs to sum to 1). Figure 2(b) shows how layers are connected in a typical CNN: groups of convolution, pooling and non-linearizing layers are repeated several times before 1-2 matrix-multiplication (or *fully connected*) layers are applied, ended by re-scaling.

The matrix multiplication and convolution layers are parameterized by weight arrays that are estimated from training data. The network description before training is called a *model architecture*, and the trained network with weights instantiated is a *model*. Training DNNs takes several days to weeks. Unlike recent systems work in CNNs [15, 17], we are not concerned here with the efficient *learning* of CNNs, but rather, their efficient *execution*. Most CNNs today tend to be linear [30, 44], but DAGs [45] and loopy graphs, known as Recurrent Neural Networks (RNNs) [27], which are fully unrolled over incoming data before execution, have also been considered. We focus on linear networks below, but our techniques are applicable to others.

Figure 3 shows resource usage per layer for AlexNet, a representative architecture originally proposed for object recognition, but now used as the basis for other tasks such a scene recognition as well. For each layer on the x-axis, the y-axis (note the log scale) reports the number of operations to execute the layer (labeled "compute"), the number of floats to represent the layer ("storage") and the number of intermediate floats generated by the layer ("output size"). Two points are key to optimizing resource usage:

1. Memory use is dominated by the weight matrix of the matrix multiplication layers (labeled "fc" for fully connected). Convolutional kernels and even intermediate data size are small in comparison.
2. Computational overhead is dominated by convolution operations, exceeding matrix multiplication overhead by an order of magnitude, with negligible overheads for pooling and non-linearization.

3.2 Model optimization

Recent *model optimization* techniques for reducing resource use in DNNs have targeted these opportunities in three main ways:

1. *Matrix factorization* replaces the weight matrices and convolution kernels by their low-rank approximations[3] [25, 28, 42, 50]. Replacing a size-$M \times M$ weight matrix $W_{M\times M}$ with its singular value decomposition $U_{M\times k}V_{k\times M}$ reduces storage overhead from M^2 to $2Mk$ and computational overhead from M^3 to $2M^2k$. The

[3]Convolution is implemented as matrix multiplication [10].

most recent results [28] have reported reductions of 5.5× in memory use and 2.7× in FLOPs at a loss of 1.7% accuracy for the "AlexNet" model we use for scene recognition and 1.2×, 4.9× and 0.5% for the "VGG16" model we use for object recognition.

2. *Matrix pruning* [13, 22] sparsifies matrices by zeroing very small values, use low-bitwidth representations for remaining non-zero values and use compressed (e.g., Huffman coded) representations for these values. The most recent results [22] report 11/8× reduction in model size and 3/5× reduction on FLOPs for AlexNet/VGGNet, while sacrificing essentially no accuracy. However, for instance, sacrificing 2% points of accuracy can improve memory size reduction to 20×.
3. *Architectural changes* [44] explore the space of model architectures, including varying the number of layers, size of weight matrices including kernels, etc. For instance, reducing the number of layers from 19 to ll results in a drop in accuracy of 4.1% points.

A common theme across these techniques is the trading off of resource use for accuracy. When applied to the distributed, streaming, multi-programmed setting of MCDNN, several related questions present themselves. What is the "best" level of approximation for any model, at any given time, given that many other models must also execute at limited energy and dollar budgets? Where (device or cloud) should this model execute? Do the streaming and multi-programming settings present opportunities for new kinds of optimizations? To clarify the issues involved, we first capture these questions in a formal problem definition below and then describe MCDNN's solution.

4. APPROXIMATE MODEL SCHEDULING

Given a stream of requests to execute models of various types, MCDNN needs to pick (and possibly generate), at each timestep, an approximate variant of these models and a location (device or cloud) to execute it in. These choices must satisfy both long-term (e.g., day-long) budget constraints regarding total energy and dollar budgets over many requests, and short-term capacity constraints (e.g., memory and processing-cycle availability). We call this problem the *approximate model scheduling (AMS)* problem and formalize it below.

4.1 Fractional packing and paging

In formalizing AMS, we are guided by the literature on online paging and packing. Our problem may be viewed as a distributed combination of online fractional packing and paging problems.

The standard fractional packing problem [8] seeks to *maximize* a linear sum of variables while allowing upper bounds on other linear sums of the same variables. In AMS, the assumption is that when some model is requested to be executed at a time step, we have the option of choosing to execute an arbitrary "fraction" of that model. In practice, this fraction is a variant of the model that has accuracy and resource use that are fractional with respect to the "best" model. These fractions x_t at each time step t are our variables. Assuming for the moment a linear relation between model fraction and accuracy, we have average accuracy over all time steps proportional to $\sum_t a_t x_t$, where a_t is the accuracy of the best model for request t. Finally, suppose one-time resource-use cost (e.g. energy, cloud cost) is proportional to the fraction of the model used, with e_t the resource use of the best variant of the model requested at t. A resource budget E over all time steps gives the upper-bound constraint $\sum_t e_t x_t \leq E$. Maximizing average accuracy under budget constraints of this form gives a packing problem.

The weighted paging problem [7] addresses a sequence of t requests, each for one of M pages. Let $x_{mj} \in [0,1]$ indicate what fraction of page m was evicted in between its ith and $i+1$th requests. The goal is to *minimize* over t requests the total number of (weighted) paging events $\sum_{mj} c_t x_{mj}$. At the same time, we must ensure *at every time step* that cache capacity is not exceeded: if k is cache size and R_{mt} is the number of requests for model m up to time t and N_t is the total number of requests in this time, we require $\forall_t N_t - \sum_m x_{mR_{mt}} \leq k$. If x's are once again interpreted as fractional models and c_i represent energy costs, this formulation minimizes day-long energy costs of paging while respecting cache capacity.

Finally, new to the AMS setting, the model may be executed either on the local device or in the cloud. The two settings have different constraints (e.g., devices have serious power and memory constraints), whereas cloud execution is typically constrained by dollars and device-to-cloud communication energy.

The *online* variant of the above problems requires that optimization be performed incrementally at each time step. For instance, which model is requested at time t, and therefore the identity of coefficient e_t is only revealed in timestep t. Fraction x_t must then be computed before $t+1$ (and similarly for c_{mj}, j and x_{mj}). On the other hand the upper bound values (e.g., E) are assumed known before the first step. Recent work based on primal-dual methods has yielded algorithms for these problems that have good behavior in theory and practice.

Below, we use these packing and paging problems as the basis for precise *specification* of AMS. However, the resulting problem does not fall purely into a packing or paging category. Our solution to the problem, described in later sections, is heuristic, albeit based on insights from the theory.

Note that we do not model several possibly relevant effects. The "cache size" may vary across timesteps because, e.g., the machine is being shared with other programs. The cost/accuracy tradeoff may change with context, since models are specializable in some contexts. Sharing across models may mean that some groups of models may cost much less together than as separately. Since different apps may register different models to handle the same input type (e.g., face ID, race and gender models may all be registered to process faces), we have a "multi-paging" scenario where models are loaded in small batches. Although we do not model these effects formally, our heuristic scheduling algorithm is flexible enough to handle them.

4.2 AMS problem definition

Our goal in this section is to provide a precise specification of AMS. We seek a form that maximizes (or minimizes) an objective function under inequalities that restrict the feasible region of optimization.

Assume a device memory of size S, device energy budget E and cloud cost budget of D. Assume a set $\{M_1,...,M_n\}$ of models, where each model M_i has a set V_i of *variants* $\{M_{ij} | n \geq i \geq 1, n_i \geq j \geq 1\}$, typically generated by model optimization. For instance, M_1 may be a model for face recognition, M_2 for object recognition, and so on. Say each variant M_{ij} has size s_{ij}, device paging energy cost e_{ij}, device execution energy cost c_{ij}, device execution latency l_{ij}, cloud execution cost cost d_{ij}, and accuracy a_{ij}. We assume accuracy and paging cost vary monotonically with size. Below, we also consider a continuous version $V_i' = \{M_{ix} | n \geq i \geq 1, x \in [0,1]\}$ of variants, with corresponding costs s_{ix}, e_{ix}, c_{ix}, l_{ix}, d_{ix} and accuracy a_{ix}.

Assume an input request sequence $m_{t_1}, m_{t_2}, ..., m_{t_T}$. Subscripts $t_\tau \in \mathbb{R}$ are timestamps. We call the indexes τ's "timesteps" below. Each m_{t_τ} is a request for model M_i (for some i) at time t_τ. For each request m_{t_τ}, the system must decide *where* (device or cloud) to execute this model. If on-device, it must decide *which variant* M_{ij}, if any, of M_i to load into the cache and execute at timestep t, while also deciding which, if any, models must be evicted from the cache to make space for M_{ij}. Similarly, if on-cloud, it must decide which variant to execute. We use the variables x, y and x' to represent these actions (paging, eviction and cloud execution) as detailed below.

Let $x_{ik} \in [0,1]$ represent the variant of model M_i executed on-device when it is requested for the kth time. Note $x_{ik} = 0$ implies no on-device execution; presumably execution will be in the cloud (see below). Although in practice $x_{ik} \in V_i$, and is therefore a discrete variable, we use a continuous relaxation $x_{ik} \in [0,1]$. In this setting, we interpret x_{ik} as representing the variant $M_{ix_{ik}}$. For model M_i, let τ_{ik} be the timestep it is requested for the kth time, and $n_{i\tau}$ be the number of times it is requested up to (and including) timestep τ. For any timestep τ, let $R(\tau) = \{M_i | n_{i\tau} \geq 1\}$ be the set of models requested up to and including timestep τ.

Let $y_{i\tau} \in [0,1]$ be the fraction of the largest variant M_i^* of M_i evicted in the τ'th timestep. Let $y_i^k = \sum_{\tau=\tau_{ik}}^{\tau_{ik+1}-1} y_{i\tau}$ be the total fraction of M_i^* evicted between its kth and $k+1$th request. Note that a fraction of several models may be evicted at each timestep.

Finally, let $x'_{ik} \in [0,1]$ represent the variant of model M_i executed on the cloud when it is requested for the k-th time. We require that a given request for a model be executed either on device or on cloud, but not both, i.e., that $x'_{ik} x_{ik} = 0$ for all i,k.

Now suppose $x_{i(k-1)}$ and x_{ik} are variants selected to serve the $k-1$th and kth requests for M_i. The energy cost for paging request k is $e_{ix_{ik}}$ if $x_{i(k-1)} \neq x_{ik}$, i.e., a different variant is served than previously, or if M_i was evicted since the last request, i.e., $y_i^{k-1} > 0$. Otherwise, we hit in the cache and the serving cost is zero. Writing $\delta(x) = 0$ if $x = 0$ and 1 otherwise, the energy cost for paging is therefore $e_{ix_{ik}} \delta(|x_{i(k-1)} - x_{ik}| + y_i^{k-1})$. Adding on execution cost c, total on-device energy cost of serving a request is therefore $e_{ix_{ik}} \delta(|x_{i(k-1)} - x_{ik}| + y_i^{k-1}) + c_{ix_{ik}}$.

We can now list the constraints on an acceptable policy (i.e., assignment to x_{ik}'s and $y_{i\tau}$'s). Across all model requests, maximize aggregate accuracy of model variants served,

$$\max_{x} \sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} a_{ix_{ik}} + a_{ix'_{ik}}, \tag{1}$$

while keeping total on-device energy use within budget:

$$\sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} e_{ix_{ik}} \delta(|x_{i(k-1)} - x_{ik}| + y_i^{k-1}) + c_{ix_{ik}} \leq E, \tag{2}$$

keeping within the cloud cost budget:

$$\sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} d_{ix'_{ik}} \leq D, \tag{3}$$

and at each timestep, not exceeding cache capacity:

$$\forall_{1 \leq \tau \leq T} \sum_{\substack{M_i \in R(\tau) \\ 1 \leq k \leq n_{i\tau}}} s_{ix_{ik}} - \sum_{\substack{M_i \in R(\tau) \\ 1 \leq \tau' \leq \tau}} s_{iy_{i\tau'}} \leq S, \tag{4}$$

ensuring that the selected model variant executes fast enough:

$$\forall_{\substack{1 \leq i \leq n \\ 1 \leq k \leq n_{iT}}} l_{ix_{ik}} \leq t_{\tau_{ik}+1} - t_{\tau_{ik}}, \tag{5}$$

and ensuring various consistency conditions mentioned above:

$$\forall_{\substack{1 \leq i \leq n \\ 1 \leq k \leq n_{iT}}} 0 \leq x_{ik}, x'_{ik}, y_{ik} \leq 1 \text{ and } x_{ik} x'_{ik} = 0 \tag{6}$$

4.3 Discussion

We seek to solve the above optimization problem online. On the surface, AMS looks similar to fractional packing: we seek to maximize the weighted sum of the variables that are introduced in an online fashion, while upper-bounding various weighted combinations of these variables. If we could reduce AMS to packing, we could use the relevant primal-dual scheme [8] to solve the online variant. However, the AMS problem as stated has several key differences from the standard packing setting:

- Most fundamentally, the paging constraint (Equation 4) seeks to impose a cache capacity constraint *for every timestep*. The fractional packing problem strongly requires the number of constraints to be fixed before streaming, and therefore independent of the number of streaming elements.
- The mappings from the fractional size of the model x_{ik} to accuracy ($a_{x_{ik}}$), energy cost ($e_{x_{ik}}$), etc., are left unspecified in AMS. The standard formulation requires these to be linear, e.g, $a_{x_{ik}} = a_i x_{ik}$.
- The standard formulation requires the upper bounds to be independent of timestep. By requiring execution latency at timestep τ_{ik} to be less than the interval $t_{\tau_{ik}+1} - t_{\tau_{ik}}$, AMS introduces a time dependency on the upper bound.
- The mutual exclusion criterion (Equation 6) introduces a non-linearity over the variables.

Given the above wrinkles, it is unclear how to solve AMS with theoretical guarantees. MCDNN instead uses heuristic algorithms motivated by ideas from solutions to packing/paging problems.

Task	Description (# training images, # test images, # class)
V	VGGNet [44] on ImageNet data
A	AlexNet on ImageNet data [18] for object recognition (1.28M, 50K, 1000)
S	AlexNet on MITPlaces205 data [52] for scene recognition (2.45M, 20K, 205)
M	re-labeled S for inferring manmade/natural scenes
L	re-labeled S for inferring natural/aritificially lighting scenes
H	re-labeled S with Sun405 [49] for detecting horizons
D	DeepFaceNet replicating [46] with web-crawled face data (50K, 5K, 200)
Y	re-labeled D for age: 0-30, 30-60, 60+
G	re-labeled D for gender: M, F
R	re-labeled D for race: African American, White, Hispanic, East Asian, South Asian, Other

Table 2: Description of classification tasks.

5. SYSTEM DESIGN

Solving the AMS problem requires two main components. First, the entire notion of approximate model scheduling is predicated on the claim that the models involved allow a useful tradeoff between resource usage and accuracy. Instead of using a single hand-picked approximate variant of a model, MCDNN dynamically picks the most appropriate variant from its *model catalog*. The model catalog characterizes precisely how model optimization techniques of Section 3.2 affect the tradeoffs between accuracy and model-loading energy on device (e_{ix}; Equation 2), model-execution energy on device (c_{ix}; Equation 2), model-execution dollar cost on cloud ($d_{ix'_{ik}}$; Equation 3), model size ($s_{ix_{ik}}$; Equation 4) and model execution latency (l_{ix}; Equation 5). Below, we provide a detailed measurement-based characterization of these tradeoffs. Also, we introduce two new model optimization techniques that exploit the streaming and multi-programming setting introduced by MCDNN. The second component for solving AMS is of course an online algorithm for processing model request streams. We present a heuristically motivated scheduling algorithm below. Finally, we describe briefly the end-to-end system architecture and implementation that incorporates these components.

5.1 Model catalogs

MCDNN generates model catalogs, which are maps from versions of models to their accuracy and resource use, by applying model optimizations to DNNs at varying degrees of optimization. In this section, we focus on applying the popular factorization, pruning and architectural-change approaches described in Section 3.2. MCDNN applies the following traditional techniques:

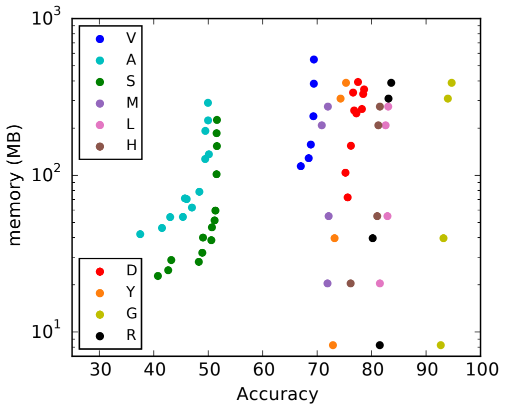


Figure 4: Memory/accuracy tradeoffs in MCDNN catalogs.

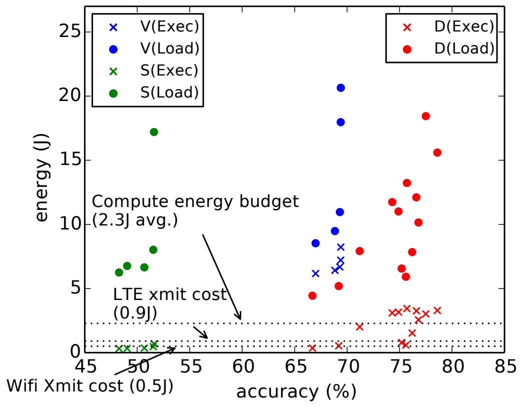


Figure 5: Energy/accuracy tradeoffs in MCDNN catalogs.

Factorization: It replaces size-$m \times n$ weight matrices with their factored variants of sizes $m \times k$ and $k \times n$ for progressively smaller values of k. It typically investigates $k = \frac{n}{2} \ldots \frac{n}{8}$. We factor both matrix multiplication and convolutional layers.

Pruning: It restricts itself to reducing the bit-widths used to represent weights in our models. We consider 32, 16 and 8-bit representations.

Architectural change: There is a wide variety of architectural transformations possible. It concentrates on the following: (1) For convolutional and locally connected layers, increase kernel/stride size or decrease number of kernels, to yield quadratic or linear reduction in computation. (2) For fully connected layers, reduce the size of the output layer to yield a linear reduction in size of the layer. (3) Eliminate convolutional layers entirely.

The individual impact of these optimizations have been reported elsewhere [22, 28, 44]. We focus here on understanding broad features of the tradeoff. For instance, does the accuracy plunge with lowering of resources or does it ramp down gently? Do various gains and trends persist across a large variety of models? How does resource use compare to key mobile budget parameters such as wireless transmission energy and commercial cloud compute costs? How do various options compare with each other: for instance, how does the energy use of model loading compare with that of execution (thus informing on the importance of caching)? Such system-level, cross-technique questions are not typically answered by the model-optimization literature.

To study these questions, we have generated catalogs for ten distinct classification *tasks* (a combination of model architecture and training data, the information a developer would input to MCDNN). We use state-of-the art architectures and standard large datasets when possible, or use similar variants if necessary. The wide variety of tasks hints at the broad utility of DNNs, and partially motivates systems like MCDNN for managing DNNs. Table 2 summarizes the classification tasks we use. For each model in the catalog, we measure average accuracy by executing it on its validation dataset, and resource use via the appropriate tool. We summarize key aspects of these catalogs below.

Figure 4 illustrates the memory/accuracy tradeoff in the MCDNN catalogs corresponding to the above classification tasks. For each classification task, we plot the set of variants produced by the model above optimization techniques in a single color. MCDNN generated 10 to 68 variants of models for various tasks. We show here points along the "Pareto curves" of the catalog i.e. the highest accuracy/lowest memory frontier. Each point in the graph illustrates the average accuracy and memory requirement of a single model variant. Note that the y-axis uses a log scale. Three points are worth noting. First, as observed elsewhere, optimization can significantly reduce resource demands at very modest accuracy loss. For instance, the VGGNet model loses roughly 4% average accuracy while using almost 10× less memory. Second, and perhaps most critically for MCDNN, accuracy loss falls off gently with resource use. If small reductions in memory use required large sacrifices in accuracy, the resource/quality tradeoff at the heart of MCDNN would be unusable. Third, these trends hold across a variety of tasks.

Figure 5 illustrates energy/accuracy tradeoffs for the face, object and scene recognition tasks. Energy numbers are measured for an NVIDIA Tegra K1 GPU on a Jetson board using an attached DC power analyzer [1]. The figure shows both execution (crosses) and load (dots) energies for each model variant. Once again, across tasks, model optimizations yield significantly better resource use and modest loss of classification accuracy. Further, the falloff is gentle, but not as favorable as for memory use. In scene recognition, for instance, accuracy falls of relatively rapidly with energy use, albeit from a very low baseline. The horizontal dotted lines indicate the energy budget (2.3J) to support 5 events/minute with 25% of a 2Ah battery, to transmit a 10kB packet over LTE (0.9J), and over WiFi (0.5J) [12]. Note the packet transmit numbers are representative numbers based on measurements by us and others, but for instance LTE packet transmit overhead may be as high as 12J for a single packet [24], and a 100ms round trip over a 700mW WiFi link could take as little as 0.07J. For the most part, standard model optimizations do not tip the balance between local execution and transmission. If execution must happen locally due to disconnection, the mobile device could support at most a few queries per minute. Finally, as the colored dots in the figure show, the energy to load models is higher by 2-5× than to execute them: reducing model loading is thus likely a key to power efficiency.

Figure 6 illustrates latency/accuracy tradeoffs for the face, object and scene recognition tasks. The figure shows time taken to execute/load models on devices (crosses) and on the cloud (dots). The measurements shown are for GPUs on-device (a Tegra K1) and on-cloud (an NVIDIA k20), and for cloud-based CPUs. The dotted lines again indicate some illuminating thresholds. For instance, a

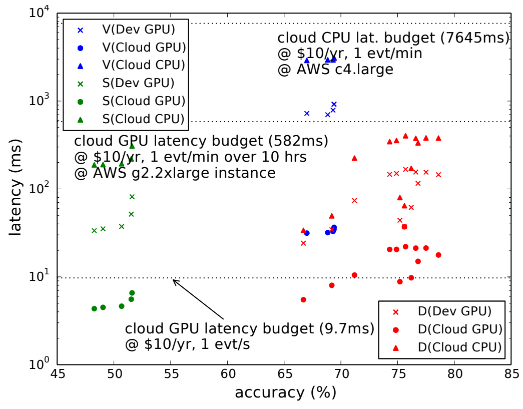


Figure 6: Latency/accuracy tradeoffs in MCDNN catalogs.

cloud-based GPU can already process one event per second on a \$10/year budget on a cloud GPU (which translates to 9.7ms per event) for scene recognition and some variants of face recognition, but not object recognition. A cloud-based CPU, however, can process one event per minute but not one per second (roughly 130ms budget, not shown) at this budget. The efficacy of model optimization carries over to execution and loading speed.

5.2 Novel model optimizations

The streaming, multi-programmed setting is a relatively uncommon one for model optimization. We therefore introduce two optimizations, one each to exploit streaming and multi-programming.

5.2.1 Specialization

One impressive ability of DNNs is their ability to classify accurately across large numbers of classes. For instance, DeepFace achieves roughly 93% accuracy over 4000 people [46]. When data flow from devices embedded in the real world, however, it is well-known that classes are heavily clustered by *context*. For instance you may tend to see the same 10 people 90% of the time you are at work, with a long tail of possible others seen infrequently; the objects you use in the living room are a small fraction of all those you use in your life; the places you visit while shopping at the mall are likewise a tiny fraction of all the places you may visit in daily life. With model specialization, MCDNN seeks to exploit class-clustering in contexts to derive more efficient DNN-based classifiers for those contexts.

We adopt a cascaded approach (Figure 7(a)) to exploit this opportunity. Intuitively, we seek to train a resource-light "specialized" variant of the developer-provided model for the few classes that dominate each context. Crucially, this model must also recognize well when an input *does not* belong to one of the classes; we refer to this class as "other" below. We chain this specialized model in series with the original "generic" variant of the model, which makes no assumptions about context, so that if the specialized model reports that an input is of class "other", the generic model can attempt to further classify it.

Figure 7(b) shows the machinery in MCDNN to support model specialization. The *profiler* maintains a cumulative distribution function (CDF) of the classes resulting from classifying inputs so far to each model. The *specializer*, which runs in the background in the cloud, determines if a small fraction of possible classes "dominate" the CDF for a given model. If so, it adds to the catalog specialized versions of the generic variants (stored in the catalog) of the model by "re-training" them on a subset of the original data dominated by these classes. If a few classes do indeed dominate strongly, we expect even smaller models, that are not particularly accurate on the general inputs, to be quite accurate on inputs drawn from the restricted context. **We seek to minimize the overhead of specialization to 10s or less, so we can exploit class skews lasting as little as five minutes, a key to making specialization broadly useful.**

Implementing the above raises three main questions. What is the criterion for whether a set of classes dominates the CDF? How can models be re-trained efficiently? How do we avoid re-training too many variants of models and focus our efforts on profitable ones? We describe how MCDNN addresses these.

The specializer determines that a CDF C is n,p*-dominated* if n of its most frequent classes account for at least fraction p of its weight. For instance, if 10 of 4000 possible people account for 90% of faces recognized, the corresponding CDF would be (10,0.9)-dominated. The specializer checks for n,p-dominance in incoming CDFs. MCDNN currently takes the simple approach of picking $n \in \{7,14,21\}$ and $p \in \{0.6,0.7,0.8,0.9,0.95\}$. Thus, for instance, if the top 7 people constitute over 60% of faces recognized, the spe-

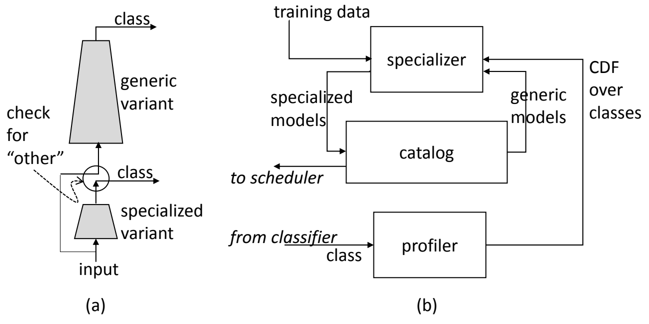


Figure 7: Model specialization: (a) Cascading specialized models. (b) MCDNN infrastructure for specialization.

cializer would add model variants to the catalog that are specialized to these seven faces.

The straightforward way to specialize a model in the catalog to a restricted context would be to re-train the schema for that model on the corresponding restricted dataset. Full retraining of DNNs is often expensive, as we discussed in the previous section. Further, the restricted datasets are often much smaller than the original ones; the reduction in data results in poorly trained models. The MCDNN specializer therefore uses a variant of the in-place transformation discussed in the previous section to retrain *just the output layer*, i.e., the last fully-connected layer and softmax layers, of the catalog model on the restricted data. We say that we *re-target* the original model. When re-training the output layer, the input values are the output of the penultimate layer, not the original training data, i.e., the input to the whole model. To obtain the output of the penultimate layer, we need flow the whole-model input through all lower layers. In an optimization we call *pre-forwarding*, we flow all training inputs through lower layers at *compile time* in order to avoid doing so at *specialization time*, an important optimization since executing lower layers cost hundreds of MFLOPs as mentioned before.

Finally, even with relatively fast re-training cost, applying it to every variant of a model and for up to $n \times p$ contexts is potentially expensive at run time. In fact, typically many of the specialized variants are strictly worse than others: they use more resources and are less accurate. To avoid this run-time expense, we use support from the MCDNN compiler. Note that the compiler cannot perform relevant specialization because the dominant classes are not known at compile time. However, for each model variant and (n,p) pair, the compiler can produce a *representative* dataset with randomly selected subsets of classes consistent with the (n,p) statistics, retarget the variant to the dataset and winnow out models that are strictly worse than others. At run time, the specializer can restrict itself to the (hopefully many fewer) remaining variant/context pairs.

5.2.2 Sharing

Until now, we have considered optimizing individual models for resource consumption. In practice, however, multiple applications could each have multiple models executing at any given time, further straining resource budgets. The *model sharing* optimization is aimed at addressing this challenge.

Figure 8(a) illustrates model sharing. Consider the case where (possibly different) applications wish to infer the identity (ID), race, age or gender of incoming faces. One option is to train one DNN for each task, thus incurring the cost of running all four simultaneously. However, recall that layers of a DNN can be viewed as increasingly less abstract layers of visual representation. It is conceivable therefore that representations captured by lower levels are shareable across many high-level tasks. If this were so, we would save the cost of

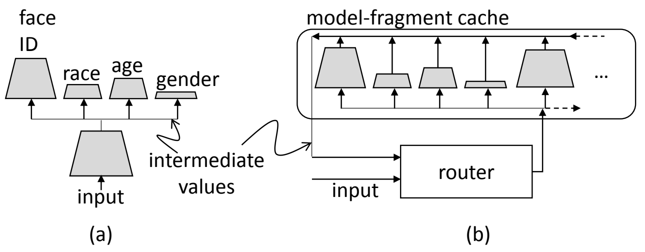


Figure 8: Model sharing: (a) Sharing model fragments for facial analysis. (b) MCDNN infrastructure for sharing, replicated in the client and cloud.

re-executing the shared bottom layers. Given that the lower (convolutional) layers of a DNN dominate its computational cost, the savings could be considerable. Indeed, we will show in the results section that re-targeting, where the shared fragment is close to the *whole* model in size is commonly applicable.

Implementing sharing requires cooperation between the MCDNN compiler and runtime. When defining input model schema, the compiler allows programmers to pick model schemas or prefixes of model schemas from a library appropriate for each domain. We currently simply use prefixes of AlexNet, VGGNet and DeepFace and their variants. Suppose the model schema s input to the MCDNN compiler has the form $s = s_l + s_u$, and t/v is the training/validation data, where layers s_l are from the library and intended for sharing. Let m_l be the trained version of s_l, also available pre-trained from the library. The compiler assembles a trained model $m_l + m_u$ consisting of two *fragments*. The lower fragment is m_l. The upper fragment, m_u, is a *freshly trained* variant of s_u, trained on $t' = m_l(t), v' = m_l(v)$, i.e., the training "input" to the upper fragment is the result of running the original training input through the lower fragment. The compiler records the fragments and their dependence in the catalog passed to the runtime. **Note that since the upper shared fragment needs to be re-trained sharing is not simply common-subexpression elimination on DNNs**.

Figure 8(b) illustrates the runtime infrastructure to support sharing. The scheduler loads complete models and model-fragments into memory for execution, notifying the router of dependencies between fragments. Given an input for classification the *router* sends it to all registered models (or their fragments). In the cases of fragments without output layers, the router collects their intermediate results and sends them on to all dependent fragments. The results of output layers are the classification results of their respective models. The router returns classification results as they become available.

5.3 MCDNN's approximate model scheduler

When applications are installed, they register with the *scheduler* a map from input types to a catalog of model fragments to be used to process those inputs, and handlers to be invoked with the result from each model. The catalog is stored on disk. When a new input appears, the scheduler (with help from the router and profiler) is responsible for identifying the model variants that need to be executed in response, paging if needed the appropriate model variants in from disk to the in-memory *model-fragment cache* in the appropriate location (i.e., on-device or on-cloud) for execution, executing the models on the input and dispatching the results to the registered handlers.

This online scheduling problem is challenging because it combines several elements considered separately in the scheduling literature. First, it has an "online paging" element [7], in that *every time an input is processed*, it must reckon with the limited capacity of the model cache. If no space is available for a model that needs to be loaded, it must evict existing models and page in new ones. Second, it has

Algorithm 1 The MCDNN scheduler.

```
1:  function PROCESS($i$,$n$)                                   ▷ $i$: input, $n$: model name
2:      if $l,m \leftarrow$ CACHELOOKUP($n$) $\neq$ null then                         ▷ Cache hit
3:          $r \leftarrow$ EXEC($m,i$)                          ▷ Classify input $i$ using model $m$
4:          async CACHEUPDATE($n,(l,m)$)                       ▷ Update cache in background
5:      else                                                                  ▷ Cache miss
6:          $m \leftarrow$ CACHEUPDATE($n$)
7:          $r \leftarrow$ EXEC($m,i$)
8:      end if
9:      return $r$
10: end function
11:
12: ▷ Update the cache by inserting the variant of $n$ most suited to current resource
    availability in the right location.
13: function CACHEUPDATE($n,(l,m) =$ nil)          ▷ Insert $n$; variant $m$ is already in
14:     $e_d,ec_s,c_c \leftarrow$ CALCPERREQUESTBUDGETS($n$)
15:     $a_d,a_s,a_c \leftarrow$ CATALOGLOOKUPRES($n,e_d,ec_s,c_c,m$)
16:     $a^*,l^* \leftarrow \max_l a_l, \mathrm{argmax}_l a_l$      ▷ Find optimal location and its accuracy
17:     $v^* \leftarrow$ CATALOGLOOKUPACC($n,a^*$)      ▷ Look up variant with accuracy $a^*$
18:     $m \leftarrow$ CACHEINSERT($l^*,v^*$) if $m.v \neq v^*$ or $l \neq l^*$ else $m$
19:     return $m$
20: end function
21:
22: ▷ Calculate energy, energy/dollar and dollar budgets for executing model $n$ on
    the mobile device, split between device/cloud and cloud only.
23: function CALCPERREQUESTBUDGETS($n$)
24:     $e,c \leftarrow$ REMAININGENERGY(),REMAININGCASH()
25:     ▷ Allocate remaining resource $r$ so more frequent requests get more resources.
    $f_i$ is the profiled frequency of model $m_i$, measured since it was last paged into
    the cache. $T$ and $r$ are the remaining time and resource budgets. $\Delta r_n$ is the cost
    to load $n$. $\Delta t_n$ is the time since $n$ was loaded, set to $\infty$ if $n$ is not in any cache.
26:     def RESPERREQ($r,l$) $= (r - \Delta r_n T/\Delta t_n) f_n / (T \Sigma_{i \in \mathrm{Cache}_l} f_i^2)$
27:     $e_d,c_d \leftarrow$ RESPERREQ($e$,"$dev$"),RESPERREQ($c$,"$cloud$")
28:     ▷ Account for split models. $t_n$ is the *fraction* of time spent executing the
    initial fragment of model $n$ relative to executing the whole.
29:     $e_s,c_s \leftarrow e_d t_n, c_d(1-t_n)$
30:     return $e_d,(e_s,c_s),c_d$
31: end function
32:
33: ▷ Find the accuracies of the model variants for model $n$ in the catalog that best
    match energy budget $e$, dollar budget $c$ and split budget $s$. If the catalog lookup is
    due to a miss in the cache (i.e., $m$ is nil, revert to a model that loads fast enough.
34: function CATALOGLOOKUPRES($n,e,s,c,m$)
35:     ▷ CLX2A($n,r$) returns accuracy of model $n$ in location $X$ using resources $r$
36:     $a_e,a_s,a_c \leftarrow$ CLD2A($n,e$),CLS2A($n,s$),CLC2A($n,c$)
37:     ▷ On a miss, bound load latency. $a_l^*$ is the accuracy of the most accurate
    model that can be loaded to location $l$ at acceptable miss latency.
38:     if $m =$ nil then
39:         $a_e,a_s,a_c \leftarrow \min(a_e^*,a_e), \min(a_s^*,a_s), \min(a_c^*,a_c)$
40:     end if
41:     return $a_e,a_s,a_c$
42: end function
43:
44: ▷ Insert variant $v$ in location $l$, where $l \in$ "$dev$","$split$","$cloud$"
45: function CACHEINSERT($l,v$)
46:     $s \leftarrow$ SIZE($v$)
47:     if ($a =$ CACHEAVAILABLESPACE($l$)) $< s$ then
48:         CACHEEVICT($l,s-a$)                 ▷ Reclaim space by evicting LRU models.
49:     end if
50:     $m \leftarrow$ CACHELOAD($l,v$)                                 ▷ Load variant $v$ to location $l$
51:     return $m$
52: end function
```

an "online packing" [8] element: *over a long period of use* (we target 10 hrs), the total energy consumed on device and the total cash spent on the cloud must not exceed battery budgets and daily cloud cost budgets. Third, it must consider processing a request either on-device, on-cloud or split across the two, introducing a *multiple-knapsack* element [9]. Finally, it must exploit the tradeoff between model accuracy and resource usage, introducing a *fractional* aspect.

It is possible to show that even a single-knapsack variant of this problem has a competitive ratio lower-bounded proportional to $\log T$, where T is the number of incoming requests. The dependency on T indicates that no very efficient solution exists. We are unaware of a formal algorithm that approaches even this ratio in the simplified setting. We present a heuristic solution here. The goal of the scheduler is to maximize the average accuracy over all requests subject to paging

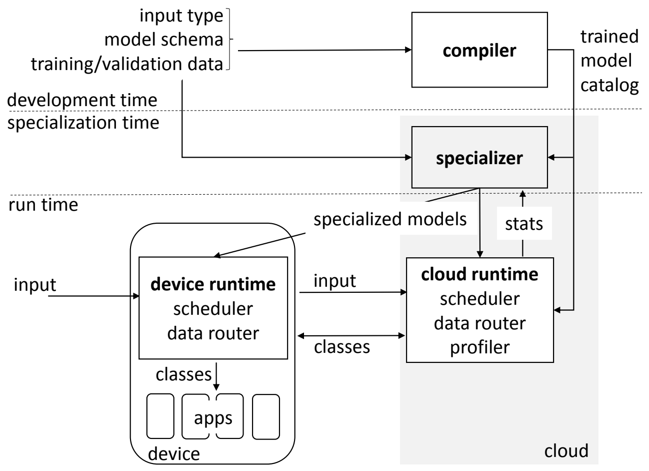


Figure 9: Architecture of the MCDNN system.

and packing constraints. The overall intuition behind our solution is to back in/out the size (or equivalently, accuracy) of a model as its use increases/decreases; the *amount* of change and the location (i.e., device/cloud/split) changed to are based on constraints imposed by the long-term budget.

Algorithm 1 provides details when processing input i on model n. On a cache miss, the key issues to be decided are *where* (device, cloud or split) to execute the model (Line 16) and *which* variant to execute (Line 17). The variant and location selected are the ones with the maximum accuracy (Line 17) under estimated future resource (energy on the device, and cash on the cloud) use (Lines 14,15).

To estimate future resource use, for model n (Lines 22-33), we maintain its frequency of use f_n, the number of times it has been requested per unit time since loading. Let us focus on how this estimation works for the on-device energy budget (Line 27) (cloud cash budgets are identical, and device/cloud split budgets (Line 29) follow easily). If e is the current *total* remaining energy budget on the device, and T the remaining runtime of the device (currently initialized to 10 hours), we allocate to n a *per-request energy budget* of $e_n = ef_n/T\Sigma_i f_i^2$, where the summation is over all models in the on-device cache. This expression ensures that *every future request* for model n is allocated energy proportional to f_n and, keeping in mind that each model i will be used Tf_i times, that the energy allocations sum to e in total (i.e., $\Sigma_i e_i f_i T = e$). To dampen oscillations in loading, we attribute a cost Δe_n to loading n. We further track the time Δt_n since the model was loaded, and estimate that if the model were reloaded at this time, and it is reloaded at this frequency in the future, it would be re-loaded $T/\Delta t_n$ times, with total re-loading cost $\Delta e_n T/\Delta t_n$. Discounting this cost from total available energy gives a refined per-request energy budget of $e_n = (e - \Delta e_n T/\Delta t_n) f_n / T\Sigma_i f_i^2$ (Line 26).

Given the estimated per-request resource budget for each location, we can consult the catalog to identify the variant providing the maximum accuracy for each location (Line 36) and update the cache at that location with that variant (Line 18). Note that even if a request hits in the cache, we consider (in the background) updating the cache for that model if a different location or variant is recommended. This has the effect of "backing in"/"backing out" models by accuracy and dynamically re-locating them: models that are used a lot (and therefore have high frequencies f_n) are replaced with more accurate variants (and vice-versa) over time at their next use.

5.4 End-to-end architecture

Figure 9 illustrates the architecture of the MCDNN system. An application developer interested in using a DNN in resource-constrained settings provides the *compiler* with the type of input to which the model should be applied (e.g., faces), a model schema, and training data. The compiler derives a *catalog* of trained models from this data, mapping each trained variant of a model to its resource costs, accuracy, and information relevant to executing them (e.g., the runtime context in which they apply). When a user installs the associated application, the catalog is stored on disk on the device and cloud and registered with the MCDNN *runtime* as handling input of a certain type for a certain app.

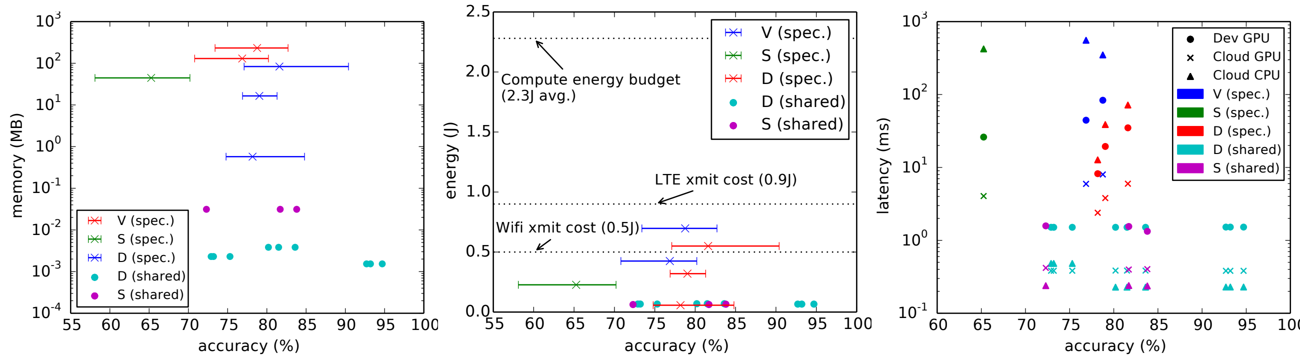
At run time, inputs for classification stream to the device. For each input, the *scheduler* selects the appropriate variant of all registered models from their catalogs, selects a location for executing them, pages them into memory if necessary, and executes them. Executing models may require *routing* data between fragments of models that are shared. After executing the model, the classification results (*classes*) are dispatched to the applications that registered to receive them. Finally, a *profiler* continuously collects *context statistics* on input data. The statistics are used occasionally by a *specializer* running in the background to specialize models to detected context, or to select model variants specialized for the context.

6. EVALUATION

We have implemented the MCDNN system end-to-end. We adapt the open source Caffe [26] DNN library for our DNN infrastructure. As our mobile device, we target the NVIDIA Jetson board TK1 board [3], which includes the NVIDIA Tegra K1 mobile GPU (with roughly 300 gflops nominal peak performance), a quad-core ARM Cortex C15 CPU, and 2GB of shared memory between CPU and GPU. The Jetson is a developer-board variant of the NVIDIA Shield tablet [4], running Ubuntu 14.04 instead of Android as the latter does. For cloud server, we use a Dell PowerEdge T620 with an NVIDIA K20c GPU (with 5GB dedicated memory and a nominal peak of 3.52 tflops), a 24-core Intel Xeon E5-2670 processors with 32 GB of memory running Ubuntu 14.04. Where cloud costs are mentioned, we use Amazon AWS G2.2xlarge single-core GPU instances and c4.large CPU instances, with pricing data obtained in early September 2015. All energy measurements mentioned are directly measured unless otherwise specified, using a Jetson board and Shield tablet instrumented with a DC power analyzer [1].

Our main results are:

- Stand-alone optimizations yield 4-10× relative improvements in memory use of models with little loss of accuracy. In absolute terms, this allows multiple DNNs to fit within mobile/embedded device memory. However, the energy used by these models, though often lower by 2× or more, is high enough that it is almost always more energy-efficient to offload execution when the device is connected to the cloud. Execution latencies on cloud CPU and GPU are also improved by a similar 2× factor, adequate to apply all but the largest models at 1 frame/minute at an annual budget of \$10, but not enough to support a realistic 1 frame/second.
- Collective optimizations (specialization and sharing), when applicable, can yield 10× to 4 orders of magnitude reductions in memory consumption and 1.2× to over 100× reductions in execution speed and energy use. These improvements have significant qualitative implications. These models can fit in a fraction of a modern phone's memory making them suitable for both consumer mobile devices and for memory-constrained embedded devices such as smart cameras. It is often less expensive to execute them locally than to offload over LTE (or sometimes even WiFi). Finally, it is feasible to process 1 frame/second on a cloud-based CPU at \$10/year.
- Both selecting which model to execute and where to run dynamically, as MCDNN does, can result in significant improvement in



(a) Memory consumption **(b)** Energy consumption **(c)** Execution latency

Figure 10: Impact of collective optimization (best viewed in color).

Task (variant)	**Time to specialize (s)**		
	Full re-train	**+ Retarget**	**+ Pre-forward**
Face (C0)	2.6e4	30.4	4.3
Face (C4)	1.4e4	24.0	4.2
Object (A0)	4.8e5	152.4	14.2
Object (A9)	9.1e4	123.0	14.1

Table 3: Runtime overhead of specialization.

the number of requests served at fixed energy and dollar budget with little loss in accuracy. Essentially, common variations in classification task mix, optimization opportunities (such a specialization and sharing) and cloud-connectivity latencies defeat static model selection and placement schemes.

- MCDNN seems well-matched with emerging applications such as virtual assistants based on wearable video and query-able personal live video streams such as Meerkat[2] or Periscope[5]. We show promising early measurements supporting these two applications.

6.1 Evaluating optimizations

Figure 10 shows the impact of *collective* optimizations, specialization and sharing. For specialization, we train and re-target progressively simpler models under increasingly favorable assumption of data skew, starting from assuming that 60% of all classes drawn belong to a group of at most size 21 (e.g., 60% of person sightings during a period all come from at most the same 21 people) to 90%/7 classes. We test these models on datasets with the same skew (e.g., 60% of faces selected from 21 people, remaining selected randomly from 1000 people). The size, execution energy consumed, and latency of the model executed under these assumptions remains fixed, only the accuracy of using the model changes, resulting in horizontal line segments in the figure. We show mean, min and max under these assumptions. Two observations are key. First, **resource usage is dramatically lower** than even the statically optimized case. For instance (Figure 10a; compare with Figure 5), specialized object recognition consumes 0.4-0.7mJ versus the stand-alone-optimized 7-16mJ. Second, **accuracy is significantly higher** than in the unspecialized setting. For instance the 0.4mJ-model has recognition rates of 70-82%, compared to the baseline 69% for the original VGGNet. Latency wins are similar. These models take less energy to run on the device than to transmit to the cloud, and easily support 1 event *per second* on a $10/year GPU budget, and *often on a CPU budget as well*.

There is no free lunch here: specialization only works when the incoming data has class skew, i.e, when a few classes dominate over the period of use of the model. Class skew may seem an onerous restriction if the skew needs to last for hours or days. Table 3 details the time required to specialize a few representative variants of models including two models for face recognition and two for object recognition (C4 is a smaller variant of C0 and A9 of A0; A0 is the standard AlexNet model for object recognition). Re-training these models from scratch takes hours to days. However, MCDNN's optimizations cut this overhead dramatically. If we only seek to re-target the model (i.e., only retrain the top layer), overhead falls to tens of seconds and pre-forwarding (see Section 5.2.1) training data through lower layers yields roughly 10-second specialization times. **MCDNN is thus well positioned to exploit data skew that lasts for as little as tens of seconds to minutes**, a capability that we believe dramatically broadens the applicability of specialization.

The benefits of sharing, when applicable, are even more striking. For scene recognition, assuming that the baseline scene recognition model (dataset S in Table 2) is running, we share all but its last layer to infer three attributes: we check if the scene is man-made or not (dataset M), whether lighting is natural or artificial (L), and whether the horizon is visible (H). Similarly, for face identification (D), we consider inferring three related attributes: age (Y), gender (G) and race (R). When sharing is feasible, the resources consumed by shared models are remarkably low: tens of kilobytes per model (cyan and purple dots in Figure 10a), roughly 100mJ of energy consumption (Figure 10b) and under 1ms of execution latency (Figure 10c), representing over 100$\times$ savings in these parameters over standard models. Shared models can very easily run on mobile devices. Put another way, inferring attributes is "almost free" given the sharing optimization.

6.2 Evaluating the runtime

Given an incoming request to execute a particular model, MCDNN decides *which variant* of the model to execute and *where* to execute it.

Selecting the right variant is especially important when executing on-device because the device has a finite-sized memory and paging models into and out of memory can be quite expensive in terms of power and latency as discussed in the previous section. MCDNN essentially pages in small variants of models and increases their size over time as their relative execution frequency increases. To gauge the benefit of this scheme, in Figure 11, we use trace-driven evaluation to compare this dynamic scheme to two schemes that page *fixed-sized* models in LRU-fashion in and out of a 600MB-sized cache. In the "Original" scheme, the models used are the unoptimized models. In the "Best" scheme, we pick models from the knees of the curves in Figure 4, so as to get the best possible accuracy-to-size tradeoff. In the "All Models" scheme, the MCDNN model picks an appropriate model variant from a catalog of model sizes.

We generate traces of model requests that result in increasing *cache load rates* with respect to the original model: the load rate is the miss-rate weighted by the size of the missed model. We generate 10 traces per load rate. Each trace has roughly 36000 requests (one

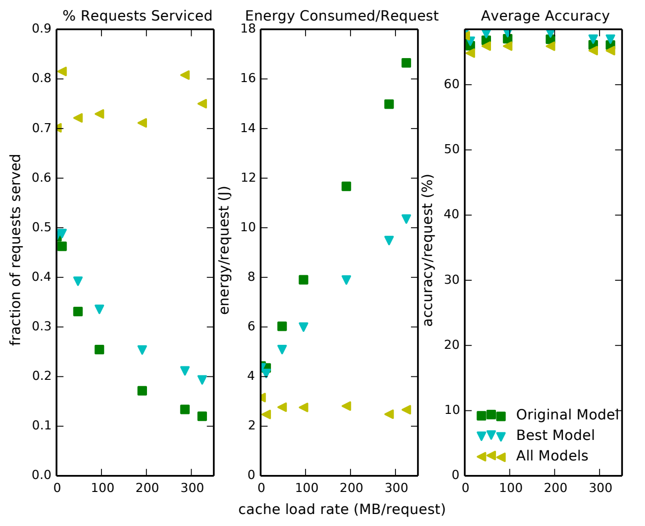


Figure 11: Impact of MCDNN's dynamically-sized caching scheme.

per second over 10 hours). Due to the energy required to service each request, none of the schemes is able to service all requests. The figure shows the number of requests served, the average energy expended per request, and the average accuracy per served request. The summary is that MCDNN constantly reduced the size of models in the cache so that even when the cache is increasingly oversubscribed with respect to fixed-size models, it still fits all the reduced-size models. As a result, MCDNN incurs minimal paging cost. **This results in a significantly higher percentage of requests served, lower average energy per request (in fact, the average energy remains fixed at execution cost for models rather than increasing amounts of paging costs), at a modest drop in accuracy**. Note that although the original and best models have higher accuracy *for requests that were served*, they serve far fewer requests.

We now move to the decision of where to execute. For each request, MCDNN uses dynamically estimated resource availability and system costs to decide where to execute. Given the relatively low cost of transmitting frames, a good strawman is a "server first" scheme that always executes on the cloud if cloud-budget and connectivity are available and on the device otherwise. This scheme if often quite good, but fails when transmission costs exceed local execution costs, which may happen due to the availability of inexpensive specialized or shared models (Figure 10b), an unexpectedly large client execution budget and/or high transmission energy costs (e.g., moving to a location far from the cloud and using a less energy-friendly transmission modality such as LTE). We examine this tradeoff in Figure 12.

Assuming full connectivity, day-long traces, and for three different transmission costs (0.1J, 0.3J and 0.5J), we examine the fraction of total requests served as increasing numbers of specialization and sharing opportunities materialize through the day. In server-first case, all requests are sent to the server, so that the number of requests served depends purely on transmission cost. **MCDNN, however, executes locally when it is less expensive to do so, thus handling a greater fraction of requests than the server-first configuration, and an increasing fraction of all requests, as specialization and sharing opportunities increase**. When transmit costs fall to 100mJ per transmit (blue line), however, remote execution is unconditionally better so that the two policies coincide.

6.3 MCDNN in applications

To understand the utility of MCDNN in practice, we built rough prototypes of two applications using it. The first, Glimpse, is a system for analyzing wearable camera input so as to provide personal assistance, essentially the scenario in Section 2. We connect the camera to an NVIDIA Shield tablet and thence to the cloud. In a production system, we assume a subsystem other than MCDNN (e.g., face detection hardware, or more generally a *gating* system [21]) that detects interesting frames and passes them on to MCDNN for classification. MCDNN then analyses these frames on the wearable and cloud. We currently use software detectors (which are themselves expensive) as a proxy for this detection sub-system. Figure 13 shows a trace reflecting MCDNN's behavior on face analysis (identification, age/race/gender recognition) and scene recognition tasks over a day[4]. The bottom panel shows the connectivity during the day. The straw man of always running on client has good accuracy but exhausts the battery. Always sending to cloud suffers during disconnected periods. MCDNN makes the trade-offs necessary to keep running through the day.

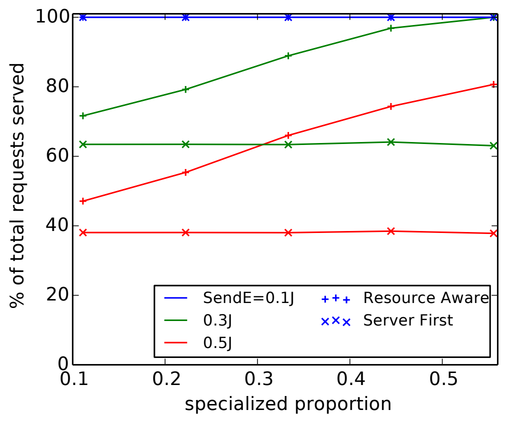


Figure 12: Impact of MCDNN's dynamic execution-location selection.

Our second application, Panorama, is a system that allows text querying of personal live video streams [2, 5] in real time. In this case, potentially millions of streams are sent to the cloud, and users can type textual queries (e.g., "man dog play") to discover streams of interest. The central challenge is to run VGGNet at 1-3 frames per second at minimal dollar cost. The key observation is that these streams have high class skew just as with wearable camera streams. We therefore aimed to apply specialized versions of VGGNet to a sample of ten 1-to-5-minute long personal video streams downloaded from YouTube; we labeled 1439 of 39012 frames with ground truth on objects present. VGGNet by itself takes an average of 364ms per frame to run on a K20 GPU at an average accuracy of 75%. MCDNN produces specialized variants of VGGNet for each video and reduces processing overhead to 42ms at an accuracy of 83%. Panorama does not run on the end-user device: MCDNN is also useful in cloud-only execution scenarios.

7. RELATED WORK

MCDNN provides shared, efficient infrastructure for mobile-cloud applications that need to process streaming data (especially video) using Deep Neural Networks (DNNs). MCDNN's main innovation is exploiting the systematic approximation of DNNs toward this goal, both revisiting scheduling to accommodate approximation, and revisiting DNN approximation techniques to exploit stream and application locality.

Recent work in the machine learning community has focused on

[4]The data fed to MCDNN in this experiment are synthesized based on statistics collected from the real world; at the time of writing we did not have Human Subjects Board permissions to capture and analyze facial data from the wild.

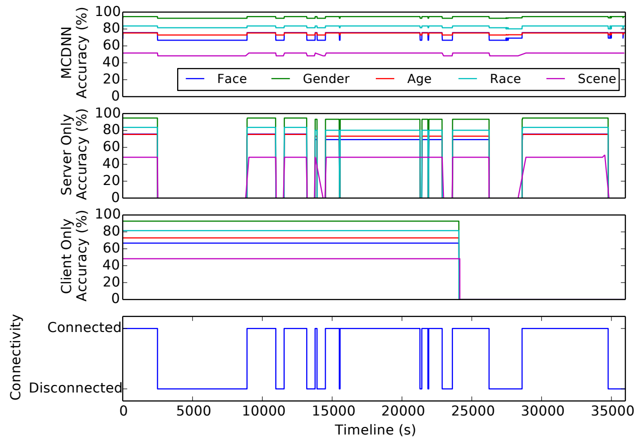


Figure 13: Accuracy of each application over time for the Glimpse usage. Each line shows a single application.

reducing the overhead of DNN execution. For important models such as speech and object recognition, research teams have spent considerable effort producing manually optimized versions of individual DNNs that are efficient at run-time [32, 42, 45]. Several recent efforts in the machine learning community have introduced automated techniques to optimize DNNs, mostly variants of matrix factorization and sparsification to reduce space [13, 20, 40, 50, 51] and computational demands [25, 28, 40, 42]. Many of these efforts support the folk wisdom that DNN accuracy can broadly be traded off for resource usage. MCDNN is complementary to these efforts, in that it is agnostic to the particular optimizations used to produce model variants that trade off execution accuracy for resources. Our primary goal is to develop a novel approximating runtime that selects between these variants while obeying various practical resource constraints over the short and long term. In doing so, we further provide both a more comprehensive set of measurements of accuracy-resource tradeoffs for DNNs than any we are aware of, and devise two novel DNN optimizations that apply in the streaming and multi-programming settings.

The hardware community has made rapid progress in developing custom application-specific integrated circuits (ASICs) to support DNNs [11, 14, 35]. We view these efforts as complementary to MCDNN, which can be viewed as a compiler and runtime framework that will benefit most underlying hardware. In particular, we believe that compiler-based automated optimization of DNNs, cross-application sharing of runtime functionality, and approximation-aware stream scheduling will all likely still be relevant and useful whenever (even very efficient) ASICs try to support multi-application, continuous streaming workloads. In the long term, however, it *is* conceivable that multiple DNNs could be run concurrently and at extremely low power on common client hardware, making system support less important.

Recent work in the mobile systems community has recognized that moving sensor-processing from the application (or library) level to middleware can help avoid application-level replication [34, 36, 43]. MCDNN may be viewed as an instance of such middleware that is specifically focused on managing DNN-based computation by using new and existing DNN optimizations to derive approximation versus resource-use tradeoffs for DNNs, and in providing a scheduler that reasons deeply about these tradeoffs. We have recently come to know of work on JouleGuard, that provides operating-system support, based on ideas from reinforcement learning control theory, for trading off energy efficiency for accuracy *with guarantees* across approximate applications that provide an accuracy/resource-use "knob" [23]. MCDNN is restricted to characterizing (and developing) such "knobs" for DNNs processing streaming workloads in particular. On the other hand, MCDNN schedules to satisfy (on a best-effort basis, with no guarantees) memory use and cloud-dollar constraints in addition to energy. MCDNN's technical approach derives from the online algorithm community [7, 8]. Understanding better how the two approaches relate is certainly of strong future interest.

Off-loading from mobile device to cloud has long been an option to handle heavyweight computations [16, 19, 38]. Although MCDNN supports both off- and on-loading, its focus is on *approximating* a *specific class* of computations (stream processing using DNNs), and on deciding automatically where to best execute them. MCDNN is useful even if model execution is completely on-client. Although we do not currently support split execution of models, criteria used by existing work to determine automatically whether and where to partition computations would be relevant to MCDNN.

Finally, we focus on the novelty of the two DNN optimizations we propose, specialization and sharing. Specialization is based on the classic technique [48] in machine-learning to structure classifiers as *cascades*. Early stages of the cascade are supposed to intercept common, easy-to-classify cases and return results without invoking more expensive later stages. Recent work has even looked into cascading DNNs [33]. All efforts to date have assumed that the class skew is evident at training time. The key difference in MCDNN's approach is that it detects class skew at test time and *dynamically* produces cascaded classifiers as appropriate. This dynamic approach dramatically broadens the settings in which cascading can be applied: we believe that dynamically-varying class skews are far more common than static ones. The machinery to detect class skew and to rapidly produce cascade layers is unique to MCDNN.

Cross-application sharing, has similarities to standard common sub-expression elimination (CSE) or memoization, in that two computations are partially unified to share a set of prefix computations at runtime. Sharing is also similar to "multi-output" DNNs trained by the machine learning community, where related classification tasks share model prefixes and are jointly trained. MCDNN's innovation may be seen as supporting multi-output DNNs in a *modular* fashion: different from jointly-trained multi-output models, the "library" model is defined and trained separately from the new model being developed. Only the upper fragment of the new model is trained subsequently. Further, to account for the fact that the (typically expensive) library model may not be available for sharing at runtime, the MCDNN compiler must train versions that do not assume the presence of a library model, and the MCDNN runtime must use the appropriate variant.

8. CONCLUSIONS

We address the problem of executing Deep Neural Networks (DNNs) on resource-constrained devices that are intermittently connected to the cloud. Our solution combines a system for optimizing DNNs that produces a catalog of variants of each model and a run-time that schedules these variants on devices and cloud so as to maximize accuracy while staying within resource bounds. We provide evidence that our optimizations can provide dramatic improvements in DNN execution efficiency, and that our run-time can sustain this performance in the face of the variation of day-to-day operating conditions.

Acknowledgments

We thank anonymous reviewers and our shepherd Deepak Ganesan for their helpful comments. We thank Tiffany Chen for helping with measurements related to the Panorama system. We thank Ishai Menache and Konstantin Makarychev for discussions and insights on the theory related to the AMS proble. This work was supported by the National Science Foundation (CNS-1318396 and CNS-1420703).

References

[1] Pwrcheck manual. http://www.westmountainradio.com/pdf/PWRcheckManual.pdf, 2015.

[2] Meerkatstreams website. http://meerkatstreams.com/, 2016.

[3] NVIDIA Jetson TK1 development board. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, 2016.

[4] NVIDIA shield website. http://shield.nvidia.com/, 2016.

[5] Twitter periscope website. http://www.periscope.tv, 2016.

[6] S. Bambach, J. M. Franchak, D. J. Crandall, and C. Yu. Detecting hands in childrens egocentric views to understand embodied attention during social interaction. *Proceedings of the Annual Meeting of the Cognitive Science Society (CogSci)*, pages 134–139, 2014.

[7] N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. *Journal of the ACM (JACM)*, 59(4):19, 2012.

[8] N. Buchbinder and J. Naor. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research*, 34(2):270–286, 2009.

[9] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

[10] K. Chellapilla, S. Puri, and P. Simard. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition (IWFHR)*, 2006.

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[12] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of The 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[13] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2285–2294, 2015.

[14] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, 2016.

[15] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[16] E. Cuervo, A. Balasubramanian, D. ki 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 (MobiSys)*, 2010.

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the Twenty-sixth Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[19] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[20] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[21] S. Han, R. Nandakumar, M. Philipose, A. Krishnamurthy, and D. Wetherall. GlimpseData: Towards continuous vision-based personal analytics. In *Proceedings of the 1st Workshop on Physical Analytics (WPA)*, 2014.

[22] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Proceedings of the Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1135–1143. 2015.

[23] H. Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–214, 2015.

[24] 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 (MobiSys)*, pages 225–238, 2012.

[25] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2014.

[26] 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 (MM)*, 2014.

[27] A. Karpathy, A. Joulin, and L. Fei-Fei. Deep fragment embeddings for bidirectional image-sentence mapping. In *Proceedings of the Twenty-eighth Annual Conference on Neural Information Processing Systems (NIPS)*, 2014.

[28] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[29] M. Kristan et al. The visual object tracking vot2015 challenge results. In *IEEE International Conference on Computer Vision Workshops (ICCVW) - Visual Object Tracking Challenge (VOT)*, 2015.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Twenty-sixth Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

[31] N. D. Lane, P. Georgiev, and L. Qendro. DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 283–294, 2015.

[32] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *Proceedings of the 14th Annual Conference of the International Speech Communication Association (Interspeech)*, 2013.

[33] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, 2015.

[34] R. LiKamWa and L. Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 213–226, 2015.

[35] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Diannaoyu: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[36] S. Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 29–42, 2012.

[37] H. Pirsiavash and D. Ramanan. Detecting activities of daily living in first-person camera views. In *Computer Vision and Pattern Recognition (CVPR)*, pages 2847–2854, 2012.

[38] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[39] S. Rallapalli, A. Ganesan, K. Chintalapudi, V. N. Padmanabhan, and L. Qiu. Enabling physical analytics in retail stores using smart glasses. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 115–126, 2014.

[40] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv:1603.05279*, Mar. 2016.

[41] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[42] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[43] A. A. Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: a system solution for sharing I/O between mobile systems. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 259–272, 2014.

[44] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[45] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[46] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[47] R. Tapu, B. Mocanu, and T. B. Zaharia. ALICE: A smartphone assistant used to increase the mobility of visual impaired people. *Journal of Ambient Intelligence and Smart Environments (JAISE)*, 7(5):659–678, 2015.

[48] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2001.

[49] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.

[50] D. Yu, J. Li, D. Yu, M. Seltzer, and Y. Gong. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2014.

[51] D. Yu, F. Seide, G. Li, and L. Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2012.

[52] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. Learning deep features for scene recognition using places database. In *Proceedings of the Twenty-eighth Annual Conference on Neural Information Processing Systems (NIPS)*, 2014.