MCDNN: An Execution Framework for Deep Neural Networks on Resource-Constrained Devices

Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, Arvind Krishnamurthy
University of Washington, Microsoft Research

Abstract

Deep Neural Networks (DNNs) have become the computational tool of choice for many applications relevant to mobile devices. However, given their high memory and computational demands, running them on mobile devices has required expert optimization or custom hardware. We present a framework that, given an arbitrary DNN, compiles it down to a resource-efficient variant at modest loss in accuracy. Further, we introduce novel techniques to specialize DNNs to contexts and to share resources across multiple simultaneously executing DNNs. Finally, we present a run-time system for managing the optimized models we generate and scheduling them across mobile devices and the cloud. Using the challenging continuous mobile vision domain as a case study, we show that our techniques yield very significant reductions in DNN 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 [19, 27], typically the province of servers. However, given the relevance of these applications to the mobile setting, and the potential for new ones, there is a strong case for executing DNNs on mobile devices. In this paper, we therefore present a framework for implementing DNN-based applications for (intermittently) cloud-connected mobile devices.

Recent approaches to enable DNNs on mobile devices include crafting efficient DNNs by hand [20] and devising custom co-processors for low-power execution on the phone [5]. However, these approaches still leave many challenges of practical mobile settings un-answered. Resource availability on devices may vary, often by the hour, multiple applications may wish to execute multiple DNNs, mobile data quotas and cloud server costs might make offloading prohibitive, developers may wish to deploy their own DNNs, good network connectivity may imply that the cloud is after all the best place to execute the network at a point in time, and the complexity of the classification task itself may vary over time due to the presence of context information.

To address these challenges, we design and implement a

	face [27]	**scene [29]**	**object [24]**
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: Costs versus benefits of DNNs applied to common image recognition tasks. Costs include time to train a model, memory to use it, and computation to process a single image. Benefits include very high accuracy of recognition, rivaling that of humans for object and face recognition.

system that provides machinery to not only *automatically* produce *efficient* variants of DNNs, but also to execute them *flexibly* across mobile devices and the cloud, across varying amounts of resources and in the presence of other applications using DNNs. We adapt a variety of well-known systems-optimization techniques to mitigate resource constraints. These include trading off quality of results for computing, splitting computations between client and the cloud such that communications needs are modest while ensuring that the pieces on the mobile device and the cloud satisfy resource availability constraints, sharing computations across applications to reduce overall client power use, sharing resources across users to pack the cloud efficiently, restructuring computations to trade off a resource that is available (e.g., computation) for one less so (e.g., memory), and exploiting locality of inputs (e.g., your work colleagues form a small subset of all the people you may ever see) to produce specialized solutions that consume fewer resources.

Our work leverages several key trends. First, the total computing power available on mobile devices and its power efficiency are improving dramatically, with the mobile device capable of supporting significant vision computing. This is further enhanced by the availability of very low-power gating circuitry [14] that provide energy efficient filters for detecting interesting events. Second, we embrace the convergence of the vision community on convolutional neural networks as the standard algorithmic framework for many continuous sensing tasks. By taking advantage of the structure and semantics of these computations, we are able to design techniques that transform relevant computations into semantically (approximately) equivalent versions that better address resource constraints. Third, as we are not handling generic computations, we are able to specify DNNs in a domain-specific language, statically compile them, apply a suite of *automated* optimization steps, and produces variants that implement the mitigating techniques mentioned above. Finally, a suitably designed runtime is able to take advantage of locality and sharing, the predictability of workloads,

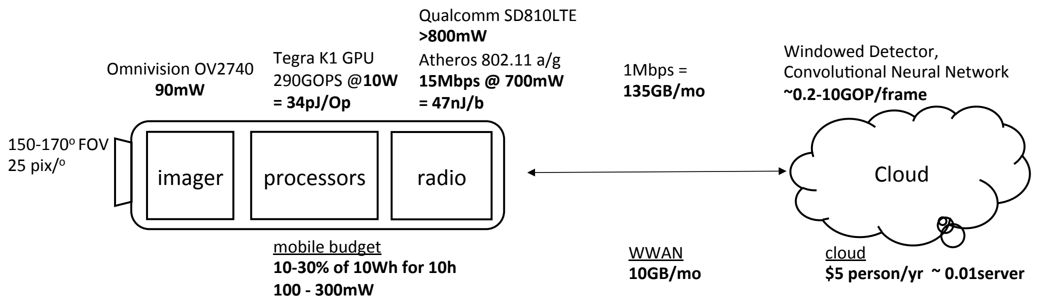


Figure 1: Basic components of a continuous mobile vision (CMV) system

and the ability to automatically factor DNNs across mobile devices and the cloud. The compiler and runtime together constitute our system, which we call MCDNN for Mobile-Cloud Deep Neural Network.

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 our dataset on very large standard datasets available from the computer vision community. Our results show that MCDNN can make effective tradeoffs between resource utilization and accuracy (e.g., transform models to use 4× fewer FLOPs and roughly 5× less memory with only a small accuracy loss), share models across DNNs with significant savings (e.g., 4-5 orders of magnitude savings in computation and memory), effectively specialize models to various contexts (e.g., specialize models with 5-25× fewer instructions, 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).

2 Continuous Mobile Vision

In this section, we introduce the continuous vision pipeline case study, paying particular attention to resource costs and budgets, and make the case that flexibility in where computations execute and the resources they consume is attractive.

The case for performing computer vision based on video streaming continuously from a wearable device has been made elsewhere [1, 13, 17, 22]. It is also common knowledge that the associated computational workloads are extremely demanding. The main response to this challenge has either been to ignore it and focus on improving the performance of recognition algorithms [12, 23] or to shift the core of the computation off the mobile device under the assumption that this workload is well beyond the capacity of the mobile device [9, 13, 22].

However, we believe that given advances in efficiency of processing and shifts in the economics of networking and cloud computing, the option to perform a large fraction (or even all) of the computer vision calculation on the mobile device is both necessary and feasible. In this context, MCDNN advocates paying the extra cost of restricting vision algorithms to those based on deep neural networks (DNNs) for the potential benefit of far more aggressive performance optimizations that make on-board execution feasible, and allows a true mobile/cloud sharing of this workload.

Figure 1 makes these challenges concrete by sketching 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 a powered computing infrastructure (some combination of a cloudlet and the deep cloud). Given the high compute demands of continuous vision, the simplest architecture is to stream video from the mobile device to the cloud, perform computer vision calculations there and send the results back. Several constraints and trends complicate this model in practice.

Network disconnection is inevitable in mobile devices. Unless applications can be designed to not use continuous vision services for long periods, it is essential to accommodate end-to-end processing on board the mobile device for extended periods. Fortunately, both the total computing power available on mobile devices and its power efficiency are improving dramatically. The latest Tegra K1-GPU based Jetson board from NVIDIA [7], for example supports 290GOPS at a 10W *whole-system-wide* power draw. Even duty cycled by 100× (yielding an average power draw of 100mW, implying that the GPU gets 10% of the 10Wh mobile battery for 10 hours), the resulting 2.9GOPS could support significant vision computing.

Even with full network connectivity, and assuming very aggressive video encoding at 1Mbps, streaming all video is prohibitive both from a mobile-energy and a wireless-bandwidth point of view. For instance, the corresponding 135GB/month is an order of magnitude more than the typical cap on customer data quota in the US. Further, keeping the radio on continuously consumes a constant 700mW, which is substantially more than the 10-30% of a generous 10Wh mobile battery that is a realistic budget for CMV applications. Combined with a low-power wide-field-of-view imager and a video encoder (a state-of-the-art Ambarella A7LW codec consumes 200mW), the costs are clearly prohibitive. Fortunately, we believe that very low-power *gating* circuitry [14] integrated with proportional-power imagers [21] will often

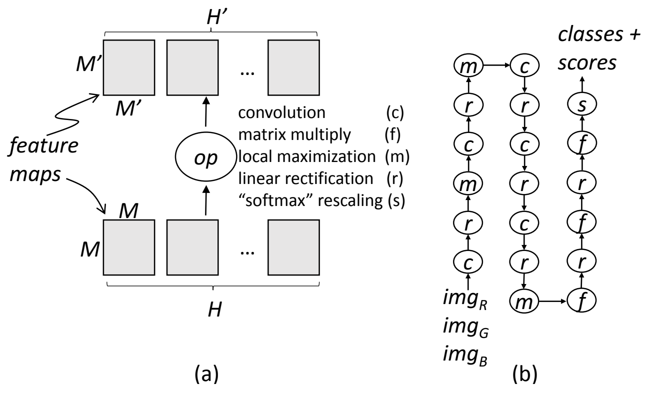


Figure 2: (a) DNN "layers" are array operations on lists of arrays called feature maps. (b) A state-of-the-art network for scene recognition, formed by connecting layers.

detect interesting events (e.g., new faces, handled objects and places) in the video at low power. For instance, face detection circuitry consumes only 30mW at 30fps [15]. We expect only 1-10% of all frames to pass this gate, so that transmitting *relevant* frames to the cloud may be feasible.

If transmitting relevant frames is within the power budget, conventional wisdom seems to favor offloading the entire visual processing of the frame to the cloud. Although full offloading at the frame level may often be the right choice, it should be weighed against two other options. First, offloading must be cheaper (from a power perspective) than full onboard processing. At 47nJ/b, a 10kB (compressed) frame will cost 3.8mJ to transmit by WiFi [25]. (WWAN numbers are similar.) The 10W NVIDIA K1-based system mentioned above will run for 0.38ms (at 290GOPS) and execute 110MOPs at this budget, possibly adequate for some vision operations. A second and perhaps less appreciated point is that cloud operators may prefer to execute as few CMV computations as possible. Note that unlike textual search queries, or even audio-based queries a la Siri, continuous vision (even at 1-10% duty cycle) entails a continuous and heavy workload per user. Given the net annual operational cost of hundreds of dollars per cloud server, a system design that runs part, most, or all of the computation on a high-performance mobile GPU *paid for and powered by the end-user* may be appealing.

The question of where best to perform (parts of) the vision computation, and how to fit these into available resources at each location, will thus vary across mobile devices, network conditions, mobile device workloads, the nature of the computation, and cost of cloud computing. Almost all these parameters vary through the day. The goal of MCDNN is to provide an easy-to-use framework that helps computations such as these fit available resources while providing flexibility in choosing where the computation occurs. The past few years have seen a dramatic shift in the computer vision community toward standardizing on DNNs, and in particular a variant called Convolutional Neural Networks (CNNs), across many key problems [24, 27, 29]. The requirement to use them is therefore not as restrictive as it may first appear.

3 Convolutional Neural Networks

We now examine convolutional neural networks (CNNs) in detail, with a view to identifying opportunities for managing their resource usage. A CNN can be viewed as a dataflow graph where the nodes, or "layers", are array-processing operations (see Figure 2). The layers are typically parameterized by weight arrays that are estimated from data using machine learning techniques. We call the network description before training a *model schema* and the trained network with weights instantiated a *model*. Unlike much recent (systems) work in CNNs [6, 10], we are not concerned here with the efficient *learning* of CNNs, but rather, their efficient *execution*. Most CNNs today tend to be linear [19, 24], but DAGs [26] and loopy graphs, known as Recurrent Neural Networks (RNNs) [18], which are fully unrolled over incoming data before execution, have also been considered. We focus on linear networks below, but our techniques carry over with little modification to DAGs and RNNs.

Each layer accepts and returns a list of arrays called *feature maps*. Table 2 provides a complete list of the types of layers used in popular networks, along with the amount of computation to execute the operation, the amount of memory required to represent the weights representing the operation, and the amount of space to represent the output of the layers. Layers are typically chained together starting from an `input` layer, and ending in a `softmax` layer, which has one output variable per class to be detected.

Some points are worth noting. First, although the CNN as a whole has complex semantics, individual layers are selected from a small number of simple, well-known, vectorizable array operations including convolution, (partial-) matrix product, local maximization, pointwise non-linearization and re-scaling (to "sum to one"). Second, the resource usage of the entire network can be estimated quite accurately by adding up usages of individual layers *in the model schema*, i.e., before the model is trained, although the accuracy of the network can only be characterized post training. Third, given that resource usage of layers is quadratic in some parameters (e.g., convolution kernel size, stride and input array size), it is possible to significantly reduce resource usage by small, judicious reductions in these parameters. Improved resource usage is typically at the cost of accuracy, and a key question is whether useful decreases in resource usage sacrifice too much accuracy. Fourth, changing parameters also reduces ouput sizes of individual layers, so that a reduction in one layer could cause cascading reductions downstream. These observations suggest that reducing the sizes of layers and retraining models may be a systematic way to reduce resource usage, at potentially lower accuracy.

Table 3 shows how these layers are composed into state-of-the-art networks for face-[27], scene-[29] and object-recognition [24]. The resource usage (to process a single image window) and accuracy figures in Table 1 are for these models. Figure 3 shows resource usage per layer: for each layer on the x-axis, the y-axis reports the number of opera-

Operation [Weight Parameters]	**Definition**	**Operations** (FLOPS)	**Rep. Size** (# floats)	**Output Size** (# floats)
`input` Pass through an $M \times M \times H$ input feature map.	$a'_{mnh} = a_{mnh}$	0	0	M^2H (same as input)
`conv[K, H', s]` Convolve incoming array by H' kernels of size $K \times K$, stride s.	For $m, n \in [0, \frac{M-K}{s})$: $a'_{mnh'} = \sum_{\substack{m',n' \in \{m,n\} \times s + \frac{K}{2} \\ 0 \le i,j < K \\ i',j' = i,j - \frac{K}{2} \\ 0 \le h < H}} a_{m'+i',n'+j',h} c_{ijhh'}$	$2((M-K)/s)^2 K^2HH'$	K^2HH'	$((M-K)/s)^2 H'$
`lconn[K, H', s]` Replace each stride-s $K \times K$ window in input array with H' uniquely weighted sums of its elements.	For $m, n \in [0, \frac{M-K}{s})$: $a'_{mnh'} = \sum_{\substack{m',n' \in \{m,n\} \times s + \frac{K}{2} \\ 0 \le i,j < K \\ i',j' = i,j - \frac{K}{2} \\ 0 \le h < H}} a_{m'+i',n'+j',h} c_{m'n'ijhh'}$	$2((M-K)/s)^2 K^2HH'$	$((M-K)/s)^2 K^2HH'$	$((M-K)/s)^2 H'$
`relu` Apply the rectified linear unit pointwise to input.	$a'_{mnh} = max(0, a_{mnh})$	0	0	M^2H (same as input)
`mpool[K, s]` Replace every stride-s $K \times K$ window in the input array with its maximum.	For $m, n \in [0, \frac{M-K}{s})$: $a'_{mnh} = \max_{\substack{m',n' \in \{m,n\} \times s + \frac{K}{2} \\ -\frac{K}{2} \le i,j < \frac{K}{2} \\ 0 \le h < H}} a_{m'+i,n'+j,h}$	$((M-K)/s)^2 K^2H$ (comparison operations)	0	$((M-K)/s)^2 H$
`fconn[M_I = M^2H, M']` Multiply size-M_I input vector by weight matrix of size $M' \times M_I$.	$A'_{M'} = F_{M'M_I} A_{M_I}$	$2M^2HM'$	M^2HM'	M'
`softmax` Re-scale inputs, representing C classes, to be in the range [0,1].	$a'_i = \frac{e^{a_i}}{\Sigma_{c=1}^{C} e^{a_c}}$	$2C$	0	1

Table 2: Cost of key DNN operations. The `fconn` layer assumes incoming feature maps are (flattened to) a vector of size $M_I = M^2H$.

tions to execute the layer, the number of floats to represent the layer and the number of floats generated by the layer. Several points are relevant to resource management.

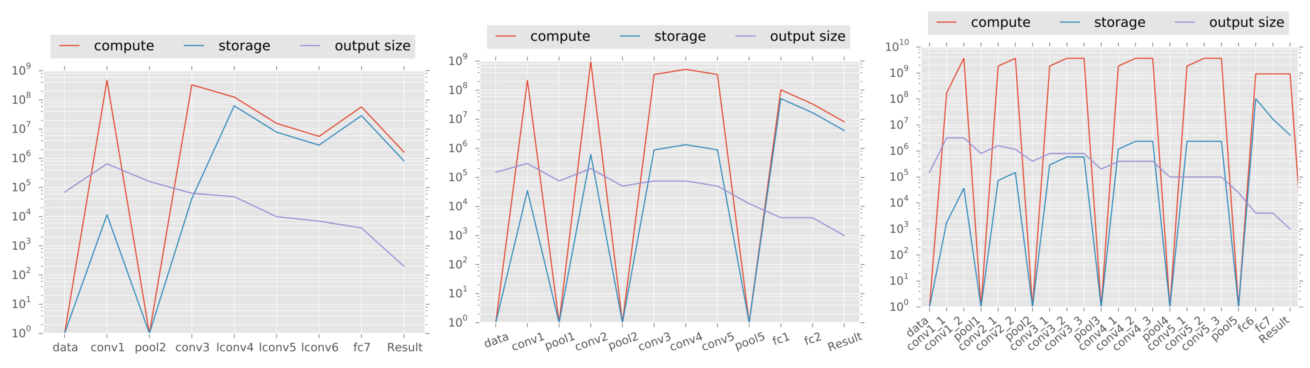
First, as noted in the introduction, trained models indeed use substantial resources (Table 1, rows 2 and 3), from 1-30 GFLOPs of processing *per image window classified* to several hundred MB of memory, motivating carefully managing their resource/quality/location tradeoff. For instance, the 110MOP-threshold identified in Section 2, below which local execution beats off-loading, seems distant.

Second, model schema (Table 3) are small, with at most tens of layers. If we wished to rewrite them in order to lower resource use, approaches that take high-polynomial or even exponential time to analyze them are plausible. On the other hand, training these schema takes days to weeks (Table 1, row 1); repeatedly training whole models is infeasible.

Finally, model structure both favors and challenges split client/cloud execution. On the one hand, the size of intermediate data passed between layers is large compared to the roughly 10kB for a *compressed* variant of the input image, and will need to be addressed carefully in a split setting where the data needs to be passed from device to cloud. On the other, the distribution of compute and memory use across layers is favorable to splitting. Convolutional layers, which appear earlier in the pipeline, are compute-heavy and memory-light, and vice-versa for fully connected layers. This supports split schemes where the early layers run on (GPU-accelerated-) devices, which are compute rich and memory poor, and later layers in the cloud where memories are large enough to keep models loaded, so that computation is the significant recurring cost.

Some aspects of CNNs relevant to resource management are not readily apparent from their resource usage patterns alone. In particular, one of the most appealing properties of deep networks is that it learns a hierarchical representation of the input domain. Layers close to the input represent low-level aspects of the input data (e.g., edges in images), whereas those progressively further away represent higher-level concepts (e.g., body parts, their relative orientation and entire faces). From an optimization viewpoint, this suggests several opportunities. First, different but related classification tasks that infer different aspects, e.g., face identification versus gender detection, of the same input, e.g., images of aces, can *share* lower layers. Second, if the classification task only changes slightly (e.g., the particular faces to be classified are different), it may be adequate to retrain just the top layer(s), avoiding very long training times. Finally, if the classification task is simplified (e.g., we only need to distinguish between a small set of faces rather than the thousands targeted by the general model), a simpler representation and smaller model may suffice.

To summarize, DNN model schema are small dataflow programs, typically comprised of tens of array-processing operations and trained on large datasets. Training is somewhat akin to whole-program compilation in traditional programs: it is slow and yields a resource-hungry "executable" or model. Familiarly, opportunities for managing execution overhead include rewriting individual operations with



(a) DeepFace **(b)** AlexNet **(c)** VGGNet

Figure 3: Resource usage of CNNs across layers (note log scale on y-axes)

DeepFaceNet (faces)	AlexNet/CNN-Places (scenes)	VGGNet (objects)
`input[152,152,3]`	`input[224,224,3]`	`input[224,224,3]`
`conv1[11,32,1]`	`conv1[11,96,4]`	`conv[3,64,1]+relu`
`relu`	`relu`	`conv[3,64,1]+relu`
`mpool1[3,2]`	`mpool1[3,2]`	`mpool[2,2]`
`conv2[9,16,1]`	`conv2[5,256,1]`	`conv[3,128,1]+relu`
`relu`	`relu`	`conv[3,128,1]+relu`
`lconn3[9,16,1]`	`mpool2[3,2]`	`mpool[2,2]`
`relu`	`conv3[3,384,1]`	`conv[3,256,1]+relu`
`lconn4[9,16,2]`	`relu`	`conv[3,256,1]+relu`
`relu`	`conv4[3,384,1]`	`conv[3,256,1]+relu`
`lconn5[7,16,1]`	`relu`	`mpool[2,2]`
`relu`	`conv5[3,256,1]`	`conv[3,512,1]+relu`
`fconn[4096]`	`relu`	`conv[3,512,1]+relu`
`relu`	`mpool5[3,2]`	`conv[3,512,1]+relu`
`fconn[4030]`	`fconn[4096]+relu`	`mpool[2,2]`
`softmax`	`fconn[4096]+relu`	`conv[3,512,1]+relu`
	`fconn[205]`	`conv[3,512,1]+relu`
	`softmax`	`conv[3,512,1]+relu`
		`mpool[2,2]`
		`fconn[4096]+relu`
		`fconn[4096]+relu`
		`fconn[1000]`
		`softmax`

Table 3: Model schema for state-of-the-art CNNs for face, scene and object recognition. We exclude M and H parameters in the definition, using values implicit from the previous layer

less expensive variants, sharing "common subexpressions" across models, specializing on runtime values and splitting across local and remote venues for execution.

DNNs, in addition, have properties that make them more amenable to optimization than traditional programs. Most fundamentally, they have a well-defined notion of approximation, which allows fully automated trade offs between accuracy and resource use for any model. Further, it is possible to predict precisely the resources a model will consume *before* it is executed. MCDNN exploits these insights by combining a "static optimizer" that produces a portfolio of variants of each model (of varying accuracy and resource use) with a run-time that seeks to use this predicted resource/accuracy tradeoff to select and schedule the optimal variant at the optimal execution location.

4 System Design

DNNs for individual classification tasks (e.g., face recognition) can, by themselves, consume a substantial fraction of the resources of a mobile device or cloud server. Given that

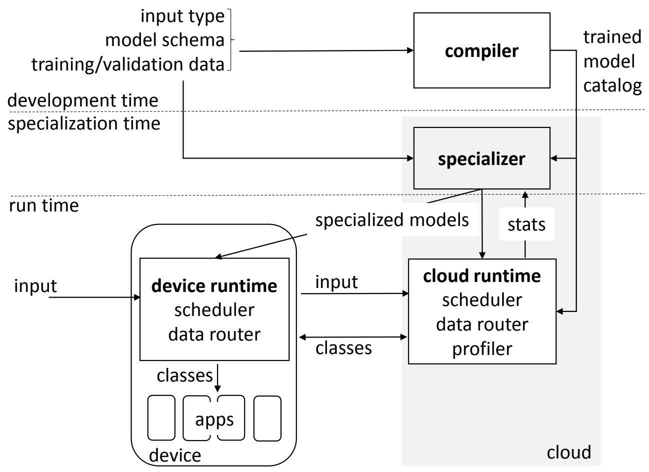


Figure 4: Architecture of the MCDNN system.

devices routinely run dozens of applications, with applications potentially registering multiple tasks, it is quite plausible that dozens of tasks will apply in parallel to incoming video, audio, touch and other input streams. For instance, not only may applications seek to identify faces, objects or scenes, they may seek to infer their attributes (e.g., whether a face is happy, a door is ajar, or a scene is outdoor). Key system resources, such as device memory, compute and communication capabilities will likely often be oversubscribed, and aggregate limits such as device battery capacities, device/cloud communication caps and cloud computing budget will be exceeded without careful management.

MCDNN uses two complementary techniques to control costs. First, it uses a variety of *model optimization* techniques to produce variants of each model that use significantly less resources than the original. Model optimization requires a combination of compile-time and run-time actions. Second, when an input requires classification, it uses *model scheduling* to select which variant to execute, paging it in if required while evicting other models, and where (device, cloud or both) to execute it, depending on availability and budgets.

Figure 4 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 handler-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 across tasks and across locations. 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. We detail these components below.

4.1 Model optimization

The fundamental premise of MCDNN is that managing many variants of a given model, with varying accuracy, resource utilization and applicability, is key to maintaining accuracy in the face of variations in resource availability. MCDNN uses four techniques to generate these variants, given a single model schema proposal and corresponding training data from developers. These techniques require a mix of compile- and run-time-support for their functioning.

4.1.1 Static transformations

Like conventional programs, DNNs can be transformed, or rewritten, to perform better. For instance, as noted in Section 3, changes in individual parameters of DNN layers can significantly reduce resource usage. For instance, the first non-input layer of the DeepFaceNet schema consumes 0.468GFLOPs (Figure 3). Since that layer is a convolution layer, Table 2 implies that doubling its stride (i.e., "re-writing" it from `conv[11,32,1]` to `conv[11,32,2]`) will reduce the number of operations by $4\times$ to 0.164GFLOPs, a substantial gain relative to the whole-network cost of 1GFLOPs.

More generally, the MCDNN compiler considers the following rewrite rules derived from Table 2. (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) Turn locally connected layers into convolution layers to yield $((M-K)/s)^2$ reduction in size of the layers, while possibly adding a small compensatory fully connected layer further along in the network. (4) Eliminate convolutional or locally connected layers.

Even considering this restricted set of rewrites, the space of possible rewrites for typical DNNs is large. We therefore further restrict ourselves to rewriting a small set of heuristically selected layers in sequence, akin to a traditional optimizing compiler. Intuitively, we seek to decrease computation use by increasing the stride of the most expensive convolutional layer(s), and to decrease memory use by reducing the output size of the largest fully connected layer(s) and by avoiding local connectivity. We consider increasing stride size and reducing the number of kernels in the two most expensive convolution layers each by a factor of 2; reducing the output vector size of the largest fully connected layer by 2, 4, 8, 16 and 32 (with much smaller layers, training is much faster); replacing individual locally connected layers with convolution layers with the possible addition of a single fully connected layer. Further, for each variant we generate, we pick up to three of these rewrites to apply.

Unlike with conventional programs, no analysis is required to ensure that the transformed network is "semantics preserving": we simply retrain it and record accuracy on validation data; bad transformations beget inaccurate networks. However, we can go about transformation in two ways. The conceptually simple approach is to transform *schemas* and retrain the resulting schemas. Alternately, adapting recent results from the machine learning community, we can incrementally transform parts of (trained) *models*. The techniques have their pros and cons and MCDNN uses both. On the one hand, retraining schemas can be extremely time consuming (e.g., recall that it can take weeks to retrain the VGGNet schemas), although the variants can be trained in parallel. On the other, the efficacy of incrementally transforming models is still poorly understood in many cases.

MCDNN uses two techniques to transform trained models. First [20], recall that fully connected layers are memory hogs because they are parameterized by dense weight arrays W of size $M' \times M$, where M and M' are sizes of their input and output vectors respectively. A simple optimization is to replace these arrays by their *low-rank matrix approximations*. Essentially we can replace W with U_k, Σ_k, and V_k of sizes $M' \times k$, $k \times k$, and $k \times M$ such that $W_k = U_k \Sigma_k V_k \approx W$ for various $k < \text{rank}(W)$. Progressively smaller values of k correspond to less accurate approximations of W (and therefore presumably of the network as a whole). The size of the new layer, however, is $k(M+M'+1)$, which is considerably less than the size MM' of W.

While the matrix approximation technique works to reduce the size of individual fully connected layers, it does not address more general rewriting. For example, suppose we wished to replace multiple convolution layers L (with schema S) with a single one L' (with schema S'). Figure 5 illustrates our generic approach to perform this transformation with only local training. We first handle the problem that the shape of the output of S' may not match that of S; we there-

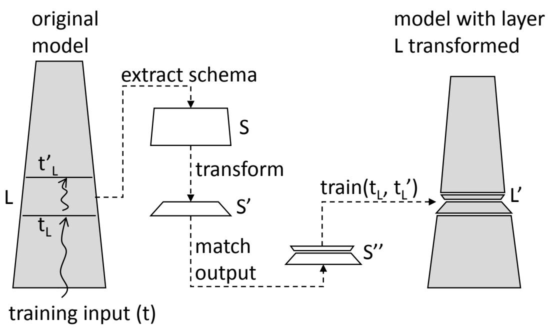


Figure 5: Transforming trained models in place.

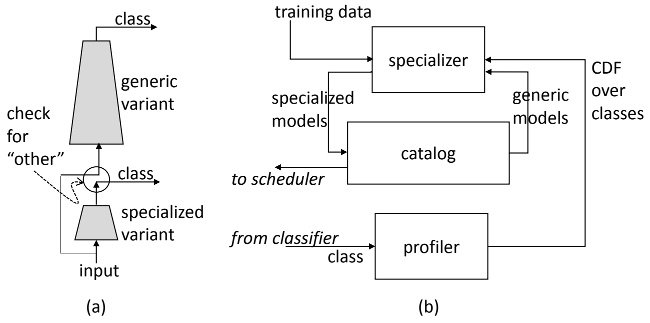


Figure 6: Model specialization: (a) Cascading specialized models. (b) MCDNN infrastructure for specialization.

fore attach to S a linear transformation layer that takes the output of S' and produces an output matching the shape of S. We are now ready to train the resulting augmented schema S''. Since we want the model L' resulting from training S'' to mimic L, we train it to input t_L (the inputs to L in the original network on training data) and to output t'_L (the outputs from L). We use the same back propagation algorithm as for training the whole network to perform this training, using a square loss function since our "mini-DNN" S'' is performing regression rather than classification.

4.1.2 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 [27]. 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 6(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 6(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.

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 below.

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 specializer 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 re-training 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.

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 com-

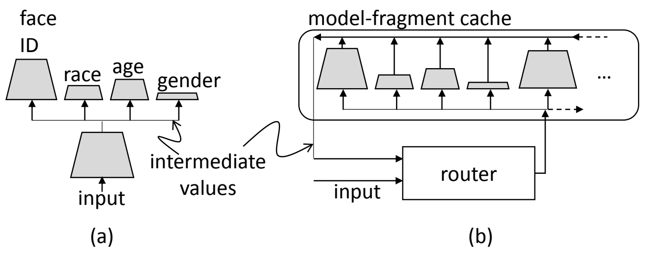


Figure 7: Model sharing: (a) Sharing model fragments for facial analysis. (b) MCDNN infrastructure for sharing, replicated in the client and cloud.

pile 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.

4.1.3 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 7(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 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.

Figure 7(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.

4.1.4 Splitting

When the connection between device and cloud is available, one option for the scheduler is to *split* models by executing one fragment on the client device and the other on the cloud. Similar to the case of sharing, for every model variant in the catalog, the compiler identifies a point at which the model could be split at run time. The broad strategy is to exploit the fact that initial layers of DNNs are computationally heavy whereas later ones are storage heavy. Further, convolution and pooling result in the size of intermediate data (i.e., the feature maps) dropping by up to an order of magnitude (c.f. Figure 3, "output size" line). The compiler writes the computational load of every layer of every model variant into the catalog. The runtime decides which layer to split at, based on the capacity of the device: it fragments the model at the last convolutional layer that fits on the device, but before the first (locally- or fully-) connected layer. The routers on the device and cloud co-ordinate to execute the two fragments sequentially and to deliver results back to the device.

4.2 Model scheduling

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, split device/cloud, 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 [2], 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 an "online packing" [3] 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 [4]. 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 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 15) and *which* variant to execute (Line 16). The variant and location selected are the ones with the maximum accuracy (Line 16) under estimated future resource (energy on the device, and cash on the cloud) use (Lines 13,14).

To estimate future resource use, for model n (Lines 20-30), 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 25) (cloud cash budgets are identical, and device/cloud split budgets (Line 27) 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 24).

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 33) and update the cache at that location with that variant (Line 17). 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.

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: function CACHEUPDATE($n, (l, m) =$ nil)     ▷ Insert $n$; variant $m$ is already in
13:     $e_d, ec_s, c_c \leftarrow$ CALCPERREQUESTBUDGETS($n$)
14:     $a_d, a_s, a_c \leftarrow$ CATALOGLOOKUPRES($n, e_d, ec_s, c_c, m$)
15:     $a^*, l^* \leftarrow \max_l a_l, \text{argmax}_l a_l$   ▷ Find optimal location and its accuracy
16:     $v^* \leftarrow$ CATALOGLOOKUPACC($n, a^*$)   ▷ Look up variant with accuracy $a^*$
17:     $m \leftarrow$ CACHEINSERT($l^*, v^*$) if $m.v \neq v^*$ or $l \neq l^*$ else $m$
18:     return $m$
19: end function
20:
21: function CALCPERREQUESTBUDGETS($n$)
22:     $e, c \leftarrow$ REMAININGENERGY(), REMAININGCASH()
23:     ▷ 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.
24:     def RESPERREQ($r, l$) $= (r - \Delta r_n T/\Delta t_n) f_n/(T\Sigma_{i \in \text{Cache}_l} f_i^2)$
25:     $e_d, c_d \leftarrow$ RESPERREQ($e$, "$dev$"), RESPERREQ($c$, "$cloud$")
26:     ▷ Account for split models. $t_n$ is the *fraction* of time spent executing the
        initial fragment of model $n$ relative to executing the whole.
27:     $e_s, c_s \leftarrow e_d t_n, c_d(1 - t_n)$
28:     return $e_d, (e_s, c_s), c_d$
29: end function
30:
31: function CATALOGLOOKUPRES($n, e, s, c, m$)
32:     ▷ CLX2A($n, r$) returns accuracy of model $n$ in location $X$ using resources $r$
33:     $a_e, a_s, a_c \leftarrow$ CLD2A($n, e$), CLS2A($n, s$), CLC2A($n, c$)
34:     ▷ 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.
35:     if $m =$ nil then
36:         $a_e, a_s, a_c \leftarrow \min(a_e^*, a_e), \min(a_s^*, a_s), \min(a_c^*, a_c)$
37:     end if
38:     return $a_e, a_s, a_c$
39: end function
40:
41: ▷ Insert variant $v$ in location $l$, where $l \in$ "$dev$", "$split$", "$cloud$"
42: function CACHEINSERT($l, v$)
43:     $s \leftarrow$ SIZE($v$)
44:     if ($a =$ CACHEAVAILABLESPACE($l$)) $< s$ then
45:         CACHEEVICT($l, s - a$)          ▷ Reclaim space by evicting LRU models.
46:     end if
47:     $m \leftarrow$ CACHELOAD($l, v$)                         ▷ Load variant $v$ to location $l$
48:     return $m$
49: end function
```

5 Evaluation

We have implemented the MCDNN system end-to-end. We adapt the open source Caffe [16] DNN library with NVIDIA's cuDNN accelerators for our DNN infrastructure. As our embedded device, we target the NVIDIA Jetson board TK1 board [7], which includes the NVIDIA Tegra K1 mobile GPU (with a 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 [8] tablet, 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.

We have assembled 11 distinct classification *tasks* (a combination of model schema and training data, the information a developer would input to MCDNN) to drive our benchmarking. We use state-of-the art schemas 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. Task A is uses the AlexNet schema on ImageNet object recognition data. The Imagenet [11] dataset is the standard for object recognition, providing 1.28M/50K images for training/validation over 1000 classes. We substitute a web-crawled face "WebFaces" dataset with 50,000/5000 training/validation images and 200 celebrity identities for the proprietary DeepFace dataset [27] from Facebook, which is much bigger, with 4M images and 4000 identities. Task C applies a carefully hand-crafted variant of DeepFaceNet, called CompactNet, aimed at better balancing resource usage with accuracy, to WebFaces. Task D applies DeepFaceNet to WebFaces. Task G applies DeepFaceNet to WebFaces re-labeled with gender labels for each face. Task R applies DeepFace to WebFaces re-labeled with race ("African American", "White", "Hispanic", "East Asian", "South Asian", "other"). Task Y applies DeepFace to WebFaces re-labeled with age ("0-30","30-60","60+"). Task S, aimed at scene recognition, applies AlexNet to the MITPlaces205 dataset (7.08M images, 205 classes) [29]. Task H applies a retargeted version of AlexNet to the MITPlaces205 dataset augmented with Sun405 [28] labels for detecting horizons in scenes. Task L uses this AlexNet/MITPlaces205/Sun405 combination with labels for inferring natural/artificially-lit scenes. Task M uses the AlexNet/MITPlaces205/Sun405 combination with labels for inferring manmade/natural scenes. Task V applies VGGNet [24] to ImageNet data.

5.1 Evaluating model optimization

We now examine closely the model catalogs generated by the MCDNN compiler given the above datasets. The goal is to understand whether the catalogs capture a useful variety of accuracy/resource datapoints, whether optimizations yield significant benefits on real data.

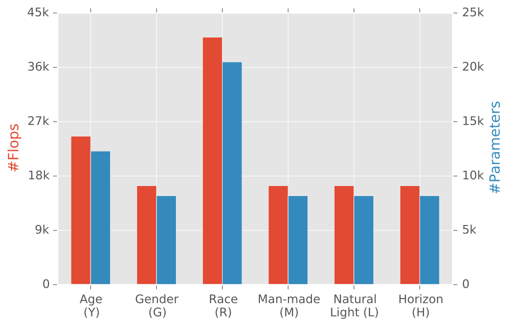
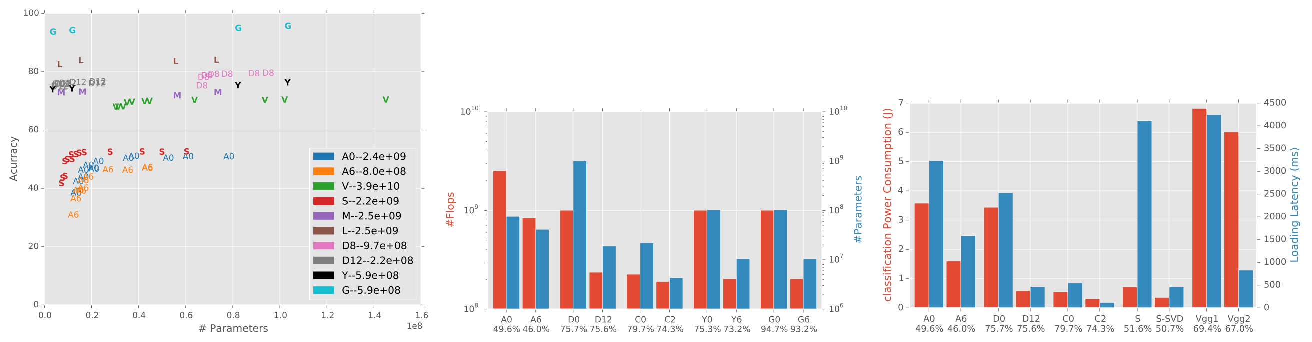


Figure 9: Incremental cost of shared models.

We applied static transformations to the input tasks A through V to generate catalogs of 10 to 68 variants for each input schema. Figure 8a shows how accuracy of a sample of the generated variants changes with resource use estimated by op- and parameter-counts (the figure shows parameter-counts). For each task, we plot the set of variants for a fixed compute cost; the legend specifies this cost in ops. Each point in the graph represents a model variant, and we use the letter designating each task as the symbol for its variants. The key point is that, **for most tasks, static transformation can reduce resource use significantly with very modest loss of accuracy**. This is critical for MCDNN: if our transformations caused accuracy to drop off precipitously with resource use, the catalog would not very useful. A related point applies to the points for tasks M, L, Y and G. These tasks use models defined for a more sophisticated task (scene and face recognition across hundreds of classes) for simpler tasks (binary classification of scemes (M,L) and up to 3- and 2-way classification of faces (Y and G)). We believe that this will be a common pattern with unsophisticated developers: they will seek to use, e.g., "the face model" for all tasks involving faces, *even if that generic model is far over-provisioned for their task*. **Static transformations are able to automatically and dramatically simplify these models while preserving high accuracy**. Finally, in most cases, **the absolute resources used by accurate variants of optimized models fall comfortably below resources available on mobile devices**: e.g., for most tasks, models with 20M parameters retain most of their accuracy. Figure 8b shows memory and compute demands for select variants of select tasks. It shows that static transformations are not restricted to improvements along a single resource dimension: they often produce models that are significantly smaller *and* faster than the input schema at modest accuracy cost.

Figure 9 illustrates the potential of sharing. We generated model variants for age, gender and race classification by sharing all but the top (classification) layer with variants of DeepFaceNet for face recognition. Similarly, we generated variants for horizon, lighting and manmade scene detection by sharing with the scene recognition model. The figure shows that the *additional* size and compute requirements are miniscule for all these variants, of the order of 10k parameters and FLOPs, relative to (hundreds of) millions for whole models. Further, as the points labeled Y, G, M and L show, these derived variants all have very good accuracy. The implication is that when basic face and scene recognition algorithms are running, **additional related inference tasks that can exploit sharing require many orders of magnitude less incremental memory and processing per additional model than without sharing**. Sharing by retargeting seems to enable almost unbounded scaling within a domain, possibly key to running large variety of tasks efficiently both on mobile device and on the cloud. In particular, it is possible to run DNN-based age, face and race (and presumably many



(a) A sample of the catalog **(b)** Size vs. compute vs. accuracy **(c)** Power and latency

Figure 8: Various views of the model catalog resulting from static transformation (best viewed in color).

Model	#Flops	#Params	FaceID	60% Context	80% Context	90% Context	95% Context	14 persons@90%	21 persons@90%
C0	225M	21.9M	79.7%	96.0/88.1%	95.3/91.0%	97.3/94.9%	97.2/96.2%	95.0%	90.8%
C1	202M	10.0M	79.6%	95.0/87.5%	95.8/90.0%	96.6/93.7%	97.2/95.8%	95.6%	91.1%
C2	190M	4.31M	74.3%	93.8/82.1%	94.3/90%	96.1/94.0%	97.2/95.8%	95.4%	90.3%
C3	183M	581K	69.2%	89.6/87.3%	91.4/85%	95.2/92.2%	96.2/94.5%	92.8%	86.2%
C4	38.6M	150K	62.1%	85.2/74.1%	86.4/78%	88.7/84.4%	89.0/87.5%	86.3%	79.7%

Table 4: Specializing the face recognition task (configuration C).

other face-related classifiers) on a mobile device simultaneously using under 2.14 floats×4B/float≊ 8.8MB of memory and kBs of incremental cost.

Table 4 evaluates the potential of context-specialization. We ask the question: if we knew that 60, 80, 90 or 95% of the people you see belong to a small group (in this case, 7 people randomly picked from the larger WebFaces dataset) and the rest are random other faces, can you use this information to produce a specialized model with higher accuracy and lower resource consumption as per Section 4.1.2? For each in-context percentage, we report numbers averaged over 5 such 7-person contexts, with roughly 350 different faces tested in each experiment (with four times that many used for training). We refer to these sub-datasets as WebFaces-Context60 through -Context95 below. We use the CompactNet model as the baseline "C0" for specialization.

We are interested in two settings. First, the "in-context-only" case, we assume the application is only interested in identifying people in context and it is adequate for all others to be reported as "other". In the other, "overall", setting we seek to identify the names of other people in the dataset as well. When Table 4 reports an accuracy as X/Y% (e.g., 96.0/86.3%), X is in-context-only accuracy whereas Y is the overall accuracy.

Columns 5-8 of the table establish that specialization still makes a sharp difference. We train specialized models for these columns by retargeting C0 to datasets WebFaces-Context60 through -Context95. The recognition accuracy of this classifier on the dataset is reported as the in-context-only result. If the classifier reports "other", we also further invoke the generic C0 to identify the nominally out-of-context face. We aggregate over the classification results to calculate the overall. In the C1-C4 rows, we generated simpler versions of C0 and specialize these simpler versions.

A few points are worth noting. If an application requires just in-context-only classification results, as the results for C3 illustrate, specialization yields roughly 90% accuracy *even if 40% of faces encountered are out of context*; moreover, the systems requires **38× less memory and 1.2× fewer FLOPs** than the highly optimized C0 model. In the entirely plausible case that 90-95% of faces seen are the same (e.g., picture an office worker in a small group of 7 or less), the last two columns of the C3 row indicate that face identification can be over 95% accurate. Finally, if higher accuracy is required, using C0 itself uniformly yields over 95% accuracy over all sub-datasets. And if somewhat lower (e.g., 85%) accuracy is tolerable, then as the C4 row shows, we require **146× less memory and 5.8× fewer FLOPs**.

If an application requires overall classification results, the table illustrates that specialization very significantly increases *overall* accuracy of classification relative to the baseline C0. On digging deeper into the data, we determined that the good performance can be explained by the fact that the in-context-only not only has good classification accuracy, it also specifically has has excellent precision and recall in recognizing the "other" class. Thus, to a first approximation, overall accuracy is simply a weighted average of accuracies of the in-context-only and unspecialized models, weighted by the in/out-of-context percentage. As the fraction of the dataset in context increases, the benefit of specialization for overall classification decreases. However, clearly **if an application is able to identify 7 or fewer classes that constitute over 60% of cases seen, specialization could yield gains of 10% or in *overall* accuracy**.

How does the context-sensitivity degrade with size of context? As per the last two columns of the table, which report in-context accuracy for models C0-C4 with 14- and 21-person contexts, the degradation is noticeable but not catastrophic, at least for contexts where 90% of test cases are in-context.

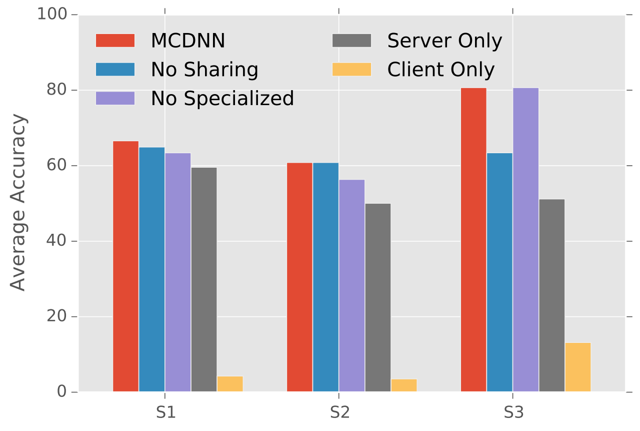


Figure 10: End-to-end performance with MCDNN versus strawmen.

	Description	Apps	Disconnect	Special
S1	Professional	D, G, R, Y, V, S	Medium	High
S2	Home maker	D, V	Rare	Medium
S3	Transit worker	D, G, R, Y, S	Often	N/A

Table 5: Description of synthesized traces. Face apps: D, G, R, Y. Object: V, Scene: S. Special column indicates probability to see the same person repeatedly.

5.2 Evaluating model scheduling

We have examined how MCDNN optimizations fare on individual inputs. We now examine how well the MCDNN scheduler handles variations in resource availability, app resource demands and input patterns. In order to test various scenarios, we synthesize input traces for three different persons usage model: a professional worker (S1), a home maker (S2), and a transit worker (S3). Each trace has different properties in the set of applications, face arrival pattern, and connectivity. For example, a transit worker sees always different people, while it is likely for a home maker to see the same faces repeatedly. Note that these traces are not representing real usage patterns, but designed to test the scheduler in various situations. The settings for each scenario is described in Table 5.

We build a simulator to run these traces over our scheduling algorithm, which makes decisions of which model to use and where to load given resource availability when each task arrives. Instead of running actual recognition, the simulator uses the average accuracy number of the used model. We run our scheduler implementing MCDNN scheduling and with four other options to compare against: two strawman systems, running always on the client, and always on the server, and two cases assuming one of features is missing, specialization and sharing. We configure energy budget as 0.5Wh for the mobile device, and \$0.0667 (\$2 for a month) as cloud budget for running 10 hours. Also we assume running a task for 1 second in a cloud server costs \$1.8e-4 according to Amazon AWS GPU server pricing.

Figure 10 depicts the average accuracy of tasks in the three traces. In S2, there is no sharable applications, and in S3, there is no specializable situation, so these features do not

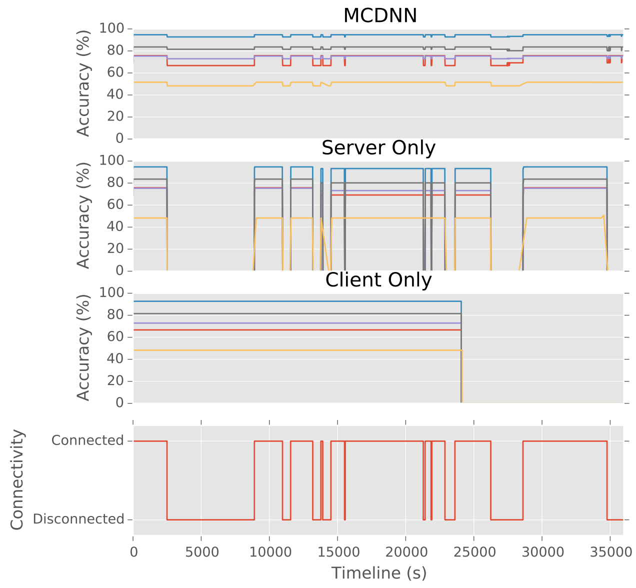


Figure 11: Accuracy of each application over time for S3. Each line shows a single application.

have effects on the result. Across all three traces, MCDNN outperforms other settings while staying in the specified energy and cost budget. To take a closer look, we show changes of accuracy for each application in Figure 11 with connectivity changes. When disconnected, the server-only scheme cannot perform recognition, thus gets 0% accuracy while MCDNN moves all tasks to the client, which causes a slight degradation to fit into the client's energy budget. For the client-only scheme, it consumes all the energy in the middle, and falls into 0% accuracy after then. In S1 and S2, there are cases when specialized models can be used—seeing the same set of people repeatedly, as MCDNN can pick the specialized models which has higher accuracy, it shows better accuracy then no-specialized mode. For no-sharing case, all face apps need to process images separately, it need to pick lower accuracy models to fit all of them, which makes it have lower average accuracy than MCDNN.

6 Conclusions

To enable efficient convolutional neural network usage on the mobile devices, we explored a variety of optimization techniques that balance the resource usage, in terms of memory and computation, and accuracy. We developed and experimented a wide range of optimized models. The results show that local optimization can achieve up to 4.9× reduction in computation and 47.7× reduction in memory with 15% loss in accuracy, which can be compensated by a combination with global optimization. Specialized context specific models can be highly compact but also with a decent accuracy. Model sharing experiments suggest that huge resources can be saved by bundling similar recognition tasks together shared with bottom layers. Further, we proposed the system design of a model compiler and specializer which can automatically optimize CNN models and create specialized compact models conforming to the resource specifications. Finally, we designed a mobile/cloud runtime system for managing and scheduling various optimized models resulting in significant performance gains.

References

[1] S. Agarwal, M. Philipose, and P. Bahl. Vision: The case for cellular small cells for cloudlets. In *International Workshop on Mobile Cloud Computing and Services*, 2014.

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

[3] N. Buchbinder and J. Naor. Online primal-dual algorithms for covering and packing. *Math. Oper. Res.*, 34(2):270–286, 2009.

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

[5] 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 *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, 2014.

[6] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014.

[7] N. Corporation. Nvidia jetson tk1 development board, 2015.

[8] N. Corporation. Nvidia shield website, 2015.

[9] 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 *MobiSys*, 2010.

[10] 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 *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012.

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

[12] A. Fathi, X. Ren, and J. M. Rehg. Learning to recognize objects in egocentric activities. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2011, Colorado Springs, CO, USA, 20-25 June 2011*, pages 3281–3288, 2011.

[13] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *MobiSys*, 2014.

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

[15] Y. Hanai, Y. Hori, J. Nishimura, and T. Kuroda. A versatile recognition processor employing Haar-like feature and cascaded classifier. In *Solid-State Circuits IEEE International Conference*, 2009.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[17] T. Kanade and M. Hebert. First-person vision. *Proceedings of the IEEE*, 100(8):2442–2453, 2012.

[18] A. Karpathy, A. Joulin, and L. Fei-Fei. Deep fragment embeddings for bidirectional image-sentence mapping. In *NIPS*, 2014.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[20] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *Interspeech*, 2013.

[21] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *MobiSys*, 2013.

[22] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Mobisys*, 2011.

[23] X. Ren and D. Ramanan. Histograms of sparse codes for object detection. In *2013 IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA, June 23-28, 2013*, pages 3246–3253, 2013.

[24] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[25] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas. Modeling wifi active power/energy consumption in smartphones. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 41–51, 2014.

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[27] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *CVPR*, 2014.

[28] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *CVPR*, 2010.

[29] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. Learning deep features for scene recognition using places database. In *NIPS*, 2014.