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

Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman
University of Washington and Microsoft Research
December 2014

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. 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 important problems in computing such as speech recognition, machine translation, handwriting recognition and many 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 [16, 17, 24], typically the province of server-class computers. However, given the relevance of the above applications to the mobile setting, and the potential for developing 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 for key tasks such as acoustic modeling [17] and devising custom coprocessors for low-power execution on the phone [4]. 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, 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. MCDNN therefore 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 resources, splitting computations between client and cloud such that communications needs are modest while ensuring that the pieces on client and 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.

In general, these techniques involve transforming relevant computations into semantically (approximately) equivalent versions that better address resource constraints. Here, the fact that we are not handling generic computations, but rather DNNs, which have relatively simple structure and semantics comes to our rescue. We assume that DNNs are specified in a domain-specific language we provide. We then show how to statically compile them, via a suite of *automated* optimization steps, into variants that implement the mitigating techniques mentioned above. Further, a suitably designed runtime selects among these variants on the basis of resource usage, locality and sharing, yielding performance that is both efficient and accurate. 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

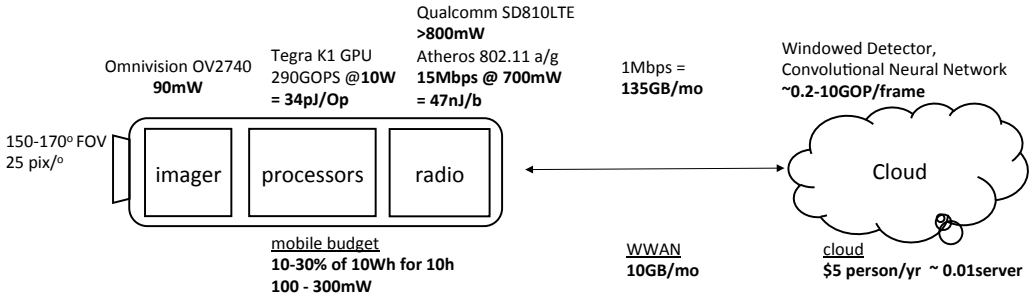


Figure 1: Basic components of a continuous mobile vision (CMV) system

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.

We make the following contributions in this paper:

- We characterize the resource demands of three widely used DNNs (for object, face, scene) in terms of memory, compute and communications demands.
- We present a novel system design comprising of a language, optimizing compiler and client/cloud runtime targeted at DNN-based mobile/cloud workloads.
- We present a design for an optimizing compiler for *individual* DNNs and demonstrate its ability. For instance, we show how to transform the well-known DeepFace model from Facebook [24] to use roughly 4× fewer FLOPs and roughly 5× less memory with only a 3.4% absolute loss in accuracy (Table 6, row 2).
- We present a novel sharing optimization *across* DNNs and show (Table 7) that it can be used to scale by many orders of magnitude the number of DNNs, including those of different types (e.g., those for face identity, gender, race and age), that can run simultaneously on a mobile device or cloud.
- We present a novel technique to automatically produce specialized DNNs that can exploit contexts commonly found in mobile devices, that are two order of magnitude more compact, use 5.5-25× fewer instructions *and* yield significantly higher accuracy than their unspecialized variants (Table 8, rows 4-5).

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 amount of resources they consume is attractive. Further, we describe a shift in the computer vision community toward standardizing on DNNs across many key problems, so that the requirement to use them is not as restrictive as it may first appear.

The case for performing computer vision based on video streaming continuously from a wearable device has been made elsewhere [1, 10, 10, 14, 19]. 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 [9, 20] 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 [6, 10, 19].

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 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, 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 [11] integrated with proportional-power imagers [18] will often 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 [12]. 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 on-board processing. At 47nJ/b, a 10kB (compressed) frame will cost 3.8mJ to transmit by WiFi [22]. (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 last three years have seen the emergence of the Deep Neural Network, and specifically the Convolution Neural Network (CNN), as the algorithm of choice for recognition problems across most fundamental vision problems [21, 24, 26]. In this paper, we use the terms DNN and CNN interchangeably. Unlike traditional approaches to computer vision where the best solutions for each problem vary broadly in computational structure, CNNs use a common architectural template and for each problem, instantiate these architectures through a combination of a declarative specification (called a *model schema*) and data-driven training. Thus, leading solutions to many of the fundamental problems in computer vision (e.g., object, scene, face and handwriting recognition) are instances of the same architectural template.

MCDNN seeks to help exploit this convergence in vision algorithms by providing an optimization suite and runtime to systematically transform CNN-based models statically and manage them at runtime so as to satisfy the constraints of the mobile/cloud setting. Of course, many mobile computer vision tasks still exist where DNNs do not form the bulk (or any) of the computational load. We view MCDNN as one of several tools to enable CMV.

3 Structure and Costs of CNNs

A convolutional neural network (CNN) can be viewed as a dataflow graph where the nodes are array-processing operations. These operations are typically parameterized by weight arrays that are estimated from data using machine learning techniques. Unlike much recent (systems) work in CNNs [5, 7], we are not concerned here with the efficient learning of CNNs, but rather, their efficient execution. Most CNNs today tend to be linear [16, 21], but DAGs [23] and loopy graphs known as Recurrent Neural Networks [15] have also been considered.

Inputs to the graph are arrays (e.g., an image or audio frame) that need to be classified or regressed. Outputs are classification or regression results; we will focus below on on classification. Each CNN node inputs and outputs a vector of arrays, which can be processed by downstream nodes. The number of operations, amount of memory consumed and size of output of a node depends on the particular array-processing operation it represents (Table 1).

A key part of MCDNN is an optimizing compiler that applies a suite of local rewrites on processing operations so as to match these counts with locally available resources. Below, we therefore discuss each of these instructions in detail, characterize commonly used CNNs in terms of these costs (Table 3) and summarize implications for continuous mobile vision applications.

3.1 CNN operations

3.1.1 `gconv`: Convolution

The convolution operation is associated with a set of H' convolution matrices (each of size $K \times K \times H$), which it applies using stride s, to its incoming array A_{MMH},

Operation	Computation (flops)	Storage (#parameters in floats)	Output size (floats)
gconv[K, H, H′, s]	$2((M-K)/s)^2K^2HH'$	K^2HH'	$((M-K)/s)^2H'$
lconv[M, K, H, H′, s]	$2((M-K)/s)^2K^2HH'$	$((M-K)/s)^2K^2HH'$	$((M-K)/s)^2H'$
inner[M, H, M′];	$2M^2HM'$	M^2HM'	M'
mpool[K, s]	$((M-K)/s)^2K^2H$ (compares)	0	$((M-K)/s)^2H$

Table 1: Cost of key DNN operations assuming input feature map array of shape $M \times M \times H$, H' output feature maps, kernels of size K with stride s and inner-product outputs of size M'

which can be thought of as H *feature map* arrays (each of size $M \times M$). Striding across $M \times M$ feature maps with stride s yields resulting feature maps A' of size $M' = \frac{M-K}{s}$, one result feature map for each of H' convolution matrices:

$$\text{gconv}_{[C_{KKHH'},s]}(A_{MMH}) = A'_{M'M'H'}$$

where, for m, n in $[0, \frac{M-K}{s})$:

$$a'_{mnh'} = \sum_{\substack{m',n' \in m,n \times s + \frac{K}{2} \\ 0 \le i,j < K \\ i',j' = i,j - \frac{K}{2} \\ 0 \le h < H}} a_{m'+i',n'+j',h} c_{ijhh'} \quad (1)$$

Note gconv requires $2 * (\frac{M-K}{s})^2 H'K^2H$ floating operations (the factor of 2 is due to the add and a multiply operations). Further, since it is parameterized by the convolution matrix $C_{KKHH'}$, it needs to store K^2HH' floating-point weights. Finally, its output $A'_{M'M'H'}$ is of size $M'^2H' = (\frac{M-K}{s})^2H'$. Given that input matrices can easily have size $M = 100$, whereas convolution matrices typically have size $K < 10$, convolution layers are high in compute requirements and low in storage requirements. However, given the relative values of stride length s and the number of feature maps H' chosen by the CNN architect, the output size may be bigger or smaller than the input.

3.1.2 lconv: Local "convolution"

The local convolution operation is identical to convolution, except that instead of sliding a single "global" convolution matrix over all positions in an incoming array, we use a distinct "local" matrix per position. Intuitively, local convolution makes sense when the incoming frame is aligned so that absolute positions within the frame have the same "meaning" every time. For example, in face recognition, since the incoming window is an aligned face, the eye, nose, mouth, chin, forehead and other parts of the face could potentially benefit from differing convolution kernels. In particular:

$$\text{lconv}_{[C_{\frac{M-K}{s}\frac{M-K}{s}KKHH',s}]}(A_{MMH}) = A'_{M'M'H'}$$

where again, for m, n in $[0, \frac{M-K}{s})$:

$$a'_{mnh'} = \sum_{\substack{m',n' \in m,n \times s + \frac{K}{2} \\ 0 \le i,j < K \\ i',j' = i,j - \frac{K}{2} \\ 0 \le h < H}} a_{m'+i',n'+j',h} c_{m'n'ijhh'} \quad (2)$$

DeepFaceNet (faces)	AlexNet/CNN-Places (scenes)	VGGNet (objects)
input[152,152,3]	input[224,224,3]	input[224,224,3]
gconv1[11,32,1]	gconv1[11,96,4]	gconv[3,64,1]+relu
relu	relu	gconv[3,64,1]+relu
mpool1[3,2]	mpool1[3,2]	mpool[2,2]
gconv2[9,16,1]	gconv2[5,256,1]	gconv[3,128,1]+relu
relu	relu	gconv[3,128,1]+relu
lconv3[9,16,1]	mpool2[3,2]	mpool[2,2]
relu	gconv3[3,384,1]	gconv[3,256,1]+relu
lconv4[9,16,2]	relu	gconv[3,256,1]+relu
relu	gconv4[3,384,1]	gconv[3,256,1]+relu
lconv5[7,16,1]	relu	mpool[2,2]
relu	gconv5[3,256,1]	gconv[3,512,1]+relu
inner[4096]	relu	gconv[3,512,1]+relu
relu	mpool5[3,2]	gconv[3,512,1]+relu
inner[4030]	inner[4096]+relu	mpool[2,2]
softmax	inner[4096]+relu	gconv[3,512,1]+relu
	inner[205]	gconv[3,512,1]+relu
	softmax	gconv[3,512,1]+relu
		mpool[2,2]
		inner[4096]+relu
		inner[4096]+relu
		inner[1000]
		softmax

Table 2: 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

The computational cost and the data output are identical for `gconv` and `lconv`. However, storage cost increases enormously (by a factor of $(\frac{M-K}{s})^2$) to accommodate the local convolution matrices.

3.1.3 `mpool`: Pooling

Pooling slides a window across the incoming window replacing the pixel at the center of the window by the maximum of its K^2 neighbors. Given that the window typically has stride s greater than one, pooling has the effect of reducing the size of its incoming feature maps by a factor of s while introducing some translational invariance to the CNN (since the maximum only needs to be in the neighborhood of the output pixel its is associated with). Pooling has no storage cost and takes $(\frac{M-K}{s})^2K^2H$ steps for the strided linear scan of its input, and result sized scaled down as in convolution:

$$\text{mpool}_{[K,s]}(A_{MMH}) = A'_{M'M'H}$$

where as usual, for m, n in $[0, \frac{M-K}{s})$:

$$a'_{mnh} = \max_{\substack{m',n' \in m, n\times s+\frac{K}{2} \\ -\frac{K}{2} \le i,j < \frac{K}{2} \\ 0 \le h < H}} a_{m'+i,n'+j,h} \tag{3}$$

3.1.4 `relu`: Non-linearization

Features that derive from strictly linear transformations of inputs are known to be limited in their representational power. The non-linearization step simply replaces every incoming feature-map element with a non-linear function of itself. Recent practice, called the Rectified Linear Unit, is to simply use a threshold minimum value, an operation that is often folded into expensive operations such as convolution and inner products and therefore has minimal incremental cost:

$$\text{relu}(a_{mnh}) = max(0, a_{mnh}) \tag{4}$$

3.1.5 `inner`: Inner product

`inner` flattens all incoming feature maps into a single high-dimensional vector A_{M_I} (where $M_I = M^2H$ if the incoming data comprises of feature maps A_{MMH}) and multiplies it by an $M' \times M_I$ weight array $F_{M'M_I}$ to produce result vector $A'_{M'}$:

$$A'_{M'} = \text{inner}_{[F_{M'M_I}]}(A_{M_I}) = F_{M'M_I}A_{M_I} \tag{5}$$

The computational cost is the $2M'M_I$ floating point addition and subtraction operations necessary for the matrix-vector product, storage cost is the M_IM' floats comprising weight-array F and output size is M' floats.

3.1.6 `softmax`: Classification

The softmax is a multi-class generalization of the logit function, and is typically the output layer of the CNN.

	DeepFaceNet	AlexNet	VGGNet
compute(flops)	1.00G	2.54G	30.9G
storage(floats)	103M	76M	138M

Table 3: Resource usage of CNNs for computer vision

The logit scales values of a single variable in $[-\infty, +\infty]$ to $[0, 1]$. Given the values of M variables as input vector a, `softmax` scales the vector so each resulting element is in $[0, 1]$ and their sums add to 1. The value of element i of its output is interpreted as the probability that the classification result of the CNN is class i:

$$\text{softmax}(a)_i = \frac{e^{a_i}}{\Sigma_{m=1}^{M} e^{a_m}} \tag{6}$$

The computational cost is that of M exponentials and their sums, with no storage costs and a result produced of size M.

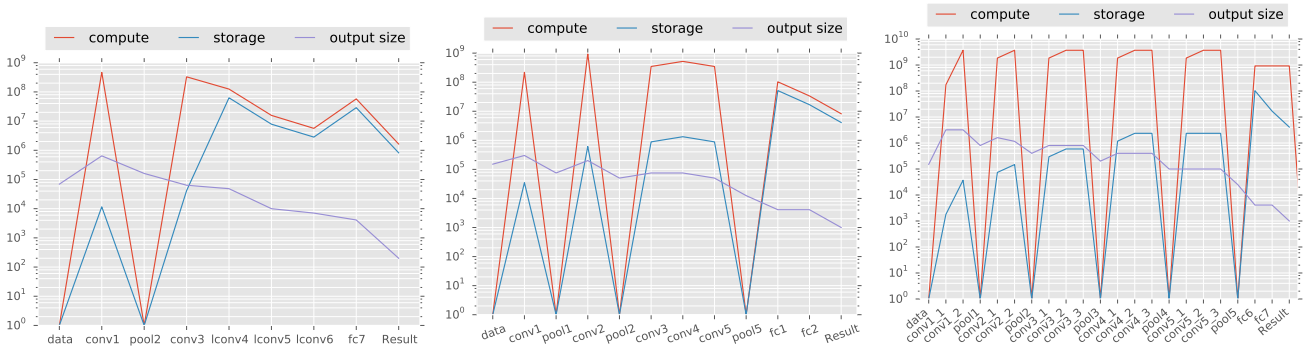
3.1.7 Summary

These operations are typically chained together starting from an `input` layer, and ending in a `softmax` layer. Table 1 summarizes the costs of the operations just discussed (we exclude `relu` and `softmax` since their costs are negligible). The summary helps expose some simple but important patterns that we will exploit later in the MCDNN compiler.

The most important point to note is that resource usage is quadratic in some parameters and linear in others. In fact, to reduce the amount of computation used in a layer, reducing incoming array size M (typically by reducing the output size from the previous layer) or kernel size K, or increasing stride s are the best bets. Reducing the number of feature maps (H and H'), for instance, has a secondary, linear effect. Finally, note that although the computational cost of pooling is relatively low (since it performs compare operations as opposed to addition and multiplication), it can have a substantial (quadratic in s) effect in reducing output size, i.e., the input size M of the next layer mentioned above.

3.2 Example CNNs

The previous section provided the detailed resource consumption of CNN operations *in terms of* their parameters. We now examine *absolute* resource consumption in practice by state-of-the-art models for face recognition [24], scene recognition [26] and object recognition [21]. All datasets involved are drawn unmodified from the internet, typically discriminate between a large number of classes (1000 objects, 4000 people and 205 scenes) use millions of images to train and tens/hundreds of thousands of images to test. Face recognition rates rival or exceed those of humans. The object recognition and scene recognition datasets have not been evaluated against humans, but anecdotal evidence suggests they are hard even for humans. Recognition rates are more modest, from 75% (for object recognition) to 53% for



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

Figure 2: Resource usage of CNNs across layers (note log scale on y-axes)

scenes, although given five guesses ("top 5"), object recognition rates shoot up to 93%. These numbers are dramatically better than those not using CNNs. It is plausible, therefore, that such networks will be a central tool in computer vision in the future.

Table 2 specifies the models. Note that although the number of CNN operations is not large, it still allows substantial experimentation in and customization of models for different domains. For instance, the DeepFace CNN, which takes aligned face images as input, uses local convolution operations, whereas the object and scene models do not.

Figure 2 illustrates their resource consumption to process *a single window in a frame* broken up across layers. Table 3 summarizes overall consumption across all layers. Each entry on the x-axes of the figure is a layer from Table 2. The y-axes represents a count of three key resources used by the corresponding layer, including the number of operations to execute the layer, the number of floats to represent the layer and the number of floats generated by the layer. Several points are worth noting:

- Overall processor and memory demands are high, of the order of a GFLOP and several hundred MB *per image window*. Given that a single frame may have multiple windows to classify (relevant objects, scenes and faces), this will strain GPU power budgets and memory budgets (assuming, e.g., a generous 10% of a 2-4GB memory budget). We certainly seem far from the 110-MOP budget (Section 2) needed to make shipping the frame to cloud redundant.
- Intermediate data sizes are fairly large compared to the roughly 10kB for a *compressed* variant of the input image: any split architecture where part of the network executes on the phone and part on the cloud will have to contend with this.
- The distribution of compute and memory use across layers is uneven. Earlier (global convolutional) layers tend to have high compute cost and low memory cost. Later (inner product) layers tend to have high memory but low compute cost. In principle, this supports schemes where the early layers are executed on the GPU-acclerated mobile device (where computation is ample, but memory tight) and later layers in the cloud where memory is more readily available and shareable across multiple clients, whereas processing entails substantial incremental cost per client. Local convolution layers are an exception to this rule: they have high memory and computation cost.

In summary, the resource consumption of CNNs is high enough that the baseline CNNs, designed with no special accommodation for resource constraints, are likely inadequate for the CMV setting.

3.3 Implications for MCDNN

The resource needs of stock CNNs from Section 3.2, when combined with the resource constraints on CMV systems from Section 2, raise several questions for a system such as MCDNN that seeks to apply CNNs in the mobile/cloud setting:

- Is it possible to reduce the resource requirements of individual models, perhaps by trading off accuracy? Are there attractive points on this trade off?
- Is it possible to reduce the aggregate requirements of multiple models (perhaps running across multiple apps), perhaps by exploiting commonality across them?
- Is it possible to structure models so that demand on cloud resources is reduced significantly while making modest demands of the mobile device?
- Is it possible to achieve above while communicating much less data than each incoming frame, perhaps on average?
- Are there settings where, even with robust connectivity, recognition on the phone is less expensive, e.g., it can be performed within 110MOPs (as calculated in Section 2) on the NVIDIA/Jetson system?

4 System Design

Figure 3 illustrates the architecture of the MCDNN system. The system has three main components, a

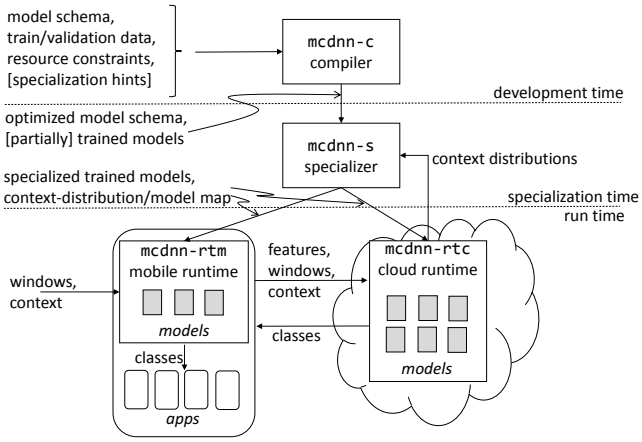


Figure 3: Architecture of the MCDNN system

compiler `mcdnn-c`, a *specializer* `mcdnn-s` and a mobile/cloud runtime collectively called `mcdnn-rt` split into a mobile component `mcdnn-rtm` and a cloud component `mcdnn-rtc`.

An application developer interested in using a DNN in resource-constrained settings provides `mcdnn-c` with the relevant model schema, data for training the model, constraints on resources to be used by these models and (optionally) a list of hints partially specifying runtime context worth specializing on, the type of data input/output from the data (e.g., a face recognition model may input windows containing faces of type $\tau_i = face$, and output face identities $\tau_o = face\_id$). The compiler derives a set of model schemas from the input schema that are optimized to meet the resource constraints, along with the models resulting from training these schemas on the input training data. When hints on runtime context are provided, the compiler also produces "partially" trained models.

When the application is installed on a particular user's device, and periodically thereafter whenever complete context information for a particular context-type is available, `mcdnn-s` is responsible for completing the training of the "partially-trained" models of that context-type using the new, complete information. These *specialized* models are intended to consume significantly less resources than their unspecialized counterparts. If no context information is available on the user at install time, MCDNN simply installs optimized models that can execute in all contexts. However, if such information is available (and as it becomes available in the future from `mcdnn-rt`, see below), the specializer will push maps from contexts to variants of models specialized to those contexts to the runtime.

Finally, at runtime, `mcdnn-rt` is responsible for selecting the most appropriate model in the client and cloud given current resource availability, application mix and context. When it detects new contexts worth specializing on, `mcdnn-rt` also invokes `mcdnn-s` to trigger the production of new specialized models.

4.1 The `mcdnn-c` compiler

Algorithm 1 Types for MCDNN

type *op_t* = ▷ DNN operations
  gconv: {kernel:int, fmaps:int, stride:int}
  —lconv: {kernel:int, fmaps:int, stride:int}
  —inner: {nelts:int}
  —...

type *schema_t* = *op_t list* ▷ DNN schemas

type *opinst_t* = ▷ Instances of DNN operations
 *—gconv_i: {k:int*int array, f:int*int*int array, s:int}*
 *—lconv_i: {k:int*int*int*int array, f:int*int*int array, s:int}*
 *—inner_i: {elts:int array * int array}*
 —...

type *model_t* = *opinst_t list* ▷ DNN models

type *hint_t* = {*nclasses: int, percent: float, ...*} ▷ Hints

type *mcmodel_t* = { ▷ MCDNN models
 schema: schema_t,
 model: model_t,
 hint: hint_t,
 accuracy: float,
 validation_set: int array list
 input_type_name(τ_i): string
 output_type_name(τ_o): string }

▷ "Less specific than" constraints
$M \prec M' \triangleq$
  $M.\tau_i = M'.\tau_i \wedge$
  $\forall i, j : M.\tau_i : M'.m(i) = M'.m(j) => M.m(i) = M.m(j)$

Algorithm 2 specifies the main tasks of `mcdnn-c`. It first invokes `reduceOpt`, which uses a series of *strictly resource-usage reducing* rewrites greedily on the incoming model schema s_0 to find new schema that fit within each of the resource constraints captured in *cs*. The results are maintained in Ms_1. Input training (t) and validation (v) data are used for training these models. Next, a post-processing step `incrOpt` that allows limited resource-usage *increasing* rewrites further optimizes these to yield models Ms_2. Next, even though specific context information about the end user is not yet available, `mcdnn-c` can use hints that partially specify runtime context to produce models Ms_3 specialized to those hints. `mcdnn-c` then looks for commonalities between these models and reference libraries of common models likely to be used by other applications at runtime to produce variants Ms_4 of the optimized models suitable for sharing. We describe each component in detail below.

Algorithm 2 `mcdnn-c`: The MCDNN compiler

1: **function** MCDNN-C($s_0, t, v, cs, hs, N, \tau = (\tau_i, \tau_o)$)
2: ▷ **input:** Model schema s_0 training (t)/validation(v) data; limits on cost of executing models cs; hints hs on class-distributions for specialization; bound N on number of greedy search steps; input/output types τ_i/τ_o.
3: ▷ **output:** MCDNN models Ms, where the model schema $M.s$ are derived from input schema s_0 to satisfy cost-constraints cs, exploit hints hs and exploit sharing opportunities implied by type constraints τ.
4: $Ms_1 = \{$REDUCEOPT(s_0, t, v, c, N, τ) **for** c **in** $cs\}$
5: $Ms_2 =$ INCROPT(Ms_1, t, v, cs, N)
6: $Ms_3 =$ PRESPECIALIZE(Ms_2, t, v, hs)
7: $Ms_4 =$ TRAINSHAREDLAYERS(Ms_3, t, v)
8: **return** Ms_4
9: **end function**

4.1.1 `reduceOpt`: Greedy cost-reducing optimization

Local optimization is based on two observations. First, in the absence of loops in deep networks, the resource usage of the network as a whole is equal to the sum of those of its component operations, *independent of run-time information*. Second, changes in individual parameters of each node can make a significant difference in resource usage. For instance, according to Figure 2, the first non-input layer of DeepFaceNet consumes 0.468GFLOPs. Since (by Table 2) that layer is a global convolution layer, we can infer from the first row, first column of Table 1 that doubling the stride on this layer (i.e., "rewriting" it from `gconv[11,32,1]` to `gconv[11,32,2]` will reduce the number of operations by a factor of 4, i.e., to 0.117GFLOPs), a substantial saving given that the entire network consumes 1GFLOP.

Algorithm 3 specifies MCDNN's greedy cost-reducing optimization stage, called `reduceOpt`. Overall, `reduceOpt` greedily rewrites the "most promising" operation in the incoming model schema s until it produces a variant of s that meets the constraints provided. The constraints include minimum accuracy a_0 of the resulting model and maximum memory (s_0) and computation (c_0) allowed by that model.

The primary risk of the a greedy scheme is that of making premature decisions that lead to local optima. We take two simple steps to mitigate this risk, and in particular to avoid overly committing to any one of our three objectives (accuracy, storage and computation) prematurely. First (Line 28) we alternate between greedily trying to reduce cost on the basis of storage and computation. Second, we avoid steps that reduce accuracy too drastically, using an externally provided threshold Δa. In order to estimate accuracy we must (Line 9), via the `train()` call) train and validate the new model schema on the user-provided data. Since some deep networks can take several days to train on today's technology [16], this is the biggest bottleneck in our compilation framework, although training time decreases rapidly as we consider smaller models.

The `transform` function finds and performs the rewrite corresponding to a single greedy step. We rely on the fact that deep network schemas are not large. `transform` therefore iterates over every parameter p and its current value v in the schema, changes the value incrementally by an externally defined increment Δv and gauges the resources used by the resulting schema. We then simply select the parameter p^* with the lowest resource usage (Line 29).

One complication is that a single parameter change can lead to cascading change in the structure of the model schema. Note, e.g., from column 3 of Table 1 that output size falls as the square of the stride s. Thus, increasing stride from 1 to 4 during our optimization reduces input sizes to the succeeding layer by a factor 16, making its output smaller, and so on. It is possible that at some layer, the input size is smaller than the convolution matrix, yielding an output size of zero. We use the `reparameterize` function to recursively prune away such layers in the graph. Note that such "cascading" effects may lead us to greedily consider catastrophically bad (from the point of view of accuracy) model schema, further motivating the accuracy-drop threshold Δa mentioned above.

4.1.2 `incrOpt`: Limited cost-increasing steps

`reduceOpt` greedily *removes* functionality from the incoming model schema so as to reduce performance demands gradually while maintaining accuracy. Further, it does so strictly by local rewrites. With the `incrOpt` step, we relax these constraints slightly. We consider rewriting multiple DNN operations simultaneously, and in particular consider *adding* functionality in a controlled manner.

Allowing non-greedy, non-local optimization can not only produce better optima in terms of total resource usage/accuracy of the DNN, it can also change *where* resources are used. In particular, consider the case where we try to split the execution of a model between mobile device and cloud. Given that DNNs usually consist of a chain of convolution operations followed by inner-product operations, they tend to be compute-heavy in the early layers and memory-heavy in the later ones. See for instance the AlexNet and VGGNet from Figure 2: convolutional layers use less than 10% of memory and execute over 80% of computations. For cloud servers, however, 300MB of memory is a much smaller bottleneck than several GOPs of processing. Splitting convolutional/inner-product calculations between mobile and cloud thus looks attractive.

In this context, depending on resource availability at the mobile end, it may be attractive to shift work from

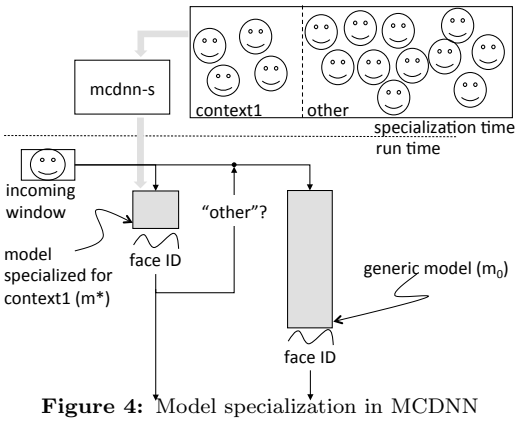


Figure 4: Model specialization in MCDNN

the convolutional layers to the inner-product layers and vice-versa. This capability becomes especially interesting for a model like DeepFaceNet, which perform *local* convolution. Local convolution layers demand high compute *and* memory resources. We therefore consider an optimization that converts local convolutional layers to global ones and introduces an inner-product layer to compensate. Less aggressively, we could rewrite convolutional layers to be less computationally demanding, but instead of sacrificing performance for it, could attempt to make it up via a relatively small amount of additional calculation in a cloud-based inner-product layer.

Our compiler currently performs a single, simple optimization on model schema output by `reduceOpt` that converts local convolution layers to global convolution and adds an additional inner-product layer. The resulting trained model is available to the run-time in case it wishes to execute the model in a split manner at any point, or if the resource-use/accuracy tradeoff is simply more favorable for the task at hand.

4.1.3 Model 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 [24]. In the mobile setting, 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 4) to exploit this opportunity. Intuitively, we seek to train a resource-light "specialized" 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 a generic resource-heavy 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.

This approach has two major unknowns. First, it is as-yet unknown whether DNNs aimed at a small subset of classes with an "other" class are much more efficient than DNNs that target a large superset. As mentioned previously, almost all layers of the DNNs are thought to compute *representations* suitable for discriminating across target classes while remaining invariant to myriad irrelevant variations in input signal, e.g., due to lighting, perspective, size, local deformations, resolution, focus, etc. How much of this computation is necessary for maintaining invariance and for discriminating against the "other" class (as opposed to discriminating between the small number of classes), as the specialized model needs to do in any case, is not well understood. In the worst case, representations to discriminate well between a few classes while remaining robust to other classes and irrelevant signal may not be much cheaper than representations for many classes. In our evaluation section, we provide the first (to our knowledge) empirical evidence that specialized deep networks can indeed be accurate *and* efficient.

Second, it is unclear how to ensure that specialization is fast and effective at runtime. If the subset of classes and contexts were known at compile time, a developer could manually explore the space of feasible model schema, train models for each schema on the data corresponding to this subset, and select the most efficient and accurate one for use in that context. However, this simple approach has two problems. First, of course, the identities of classes that cluster in contexts are *not* known until a particular user installs or, more likely, uses a program: even if the developer anticipates a "small office" scenario, they cannot possibly anticipate *who* would be in that office. It is infeasible to wait until install- or run-time to select a schema and train a model because as mentioned in Section 4.1.1, exploring the space of models to find efficient models is slow because training DNNs is slow. Second, the smaller the subset of classes in a context, the greater the potential benefit from specialization, but *the smaller the corresponding amount of training data*: for instance a subset corresponding to 5% of classes will roughly contain only 5% of the training data for all classes. DNNs are famously data hungry and there is a strong danger here of performance falling simply due to insufficient data.

To address this problem, `mcdnn-c` applies the intuition that lower layers (e.g., all but the top fully-connected and softmax layer) of DNNs compute *representations* of the domain over which they need to classify (e.g., faces, objects, indoor scenes). The top layer then

computes the classification boundary between particular classes to be recognized in this domain (e.g., particular faces, objects, etc.). The lower layers may therefore be trained on data for the entire domain *before* the particular classes to be recognized are known, with just the top "classification" layer re-trained at run-time when the specific classes to be recognized are known. We call this process of re-training just the top layer *re-targeting*.

`mcdnn-c` addresses these problems by performing much of the work of training via a pre-processing step at compile time, via the `preSpecialize` function [Algorithm 2, Line 6] and only "re-targeting", rather than training from scratch, at run time.

At compile time, hints hs partially specify the distributions of classes to be specialized for at run time. Hint $\{n : \text{int}, p : \text{float}, \ldots\}$, refers to a situation where at least fraction p of data from the top n classes to be recognized (e.g., with a face recognition DNN, $n, p = (5, 0.6)$ represents the case where at least 60% of images belong to the 5 most commonly identified people).

Let $[(m_i, M_i, a_i)]$ be the list of model schemas along with trained models and accuracies produced by . (Algorithm 2) that `mcdnn-c` uses local compilation to produce highly resource efficient candidate models that predict the original (complete) set of classes with modest accuracy. It then re-targets these models to randomly selected sets of n classes augmented with data representing an "other" class picked from classes not in the selected set. It picks the candidate model s^* that performs best on this sample data as the basis for user-specific specialization by `mcdnn-s`.

As Figure 4 shows, when the context for a particular user is known at run time, `mcdnn-s` re-targets s^* (i.e. simply retrains the softmax layer) to produce a compact, custom model. At run time, given an image window to classify, the MCDNN runtime first routes it to s^*. If its answer is "other", only then is the window routed to the presumably more expensive generic model (s_0). Given that in-context frames are by definition much more frequent than out-of-context, this should yield substantial reductions in resource consumption if s^* is cheaper than s_0. Further, s_0 may be positioned on the cloud so that we have the triple benefit of lower on-mobile compute usage, image window communication to cloud *and* use of cloud compute cycles.

4.1.4 Model sharing

Until now, we have considered the conventional setting where we optimize an individual model 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.

The underlying idea behind model sharing is that the layers of a DNN can be viewed as increasingly less abstract layers of visual representation. The bottom convolution layer captures variations in shading and different edge orientations. The very highest layers represent concepts semantically close to the entities being classified (e.g., parts of faces, objects and buildings for face, object and scene recognition). It is conceivable therefore that representations captured by lower levels are shareable across many high-level visual recognition tasks. In the limit, if the tasks can be decided using the same high-level concepts, it may be possible to re-use all the layers of one DNN across all those tasks.

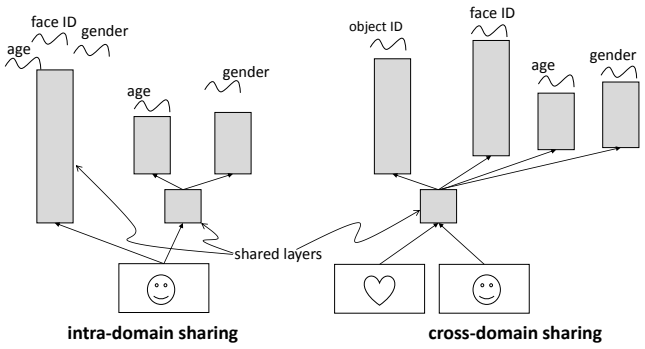


Figure 5: Model sharing in MCDNN

Figure 5 illustrates model sharing. In the *intra-domain* setting, consider the case where (possibly different) applications wish to infer the identity (ID), age and gender of incoming faces. One option is to train one DNN for each task, thus incurring the cost of running all three simultaneously. At the other extreme, using the intuition that it should be possible to determine age and gender based on the same high level features as someone's identity, we may seek to retrain just the softmax layer of the face identification DNN to determine age and gender as well (we call this technique *re-targeting*). In this case, the cost is essentially just the maximum cost of all three (ignoring the minimal cost of the softmax). Alternately, if no application required face ID, one may wish to avoid executing the relatively expensive face-ID DNN and instead execute one much simpler DNN each for gender and age where the lower layers of the two are shared (*partial sharing*). Here we would save the cost of re-executing the shared bottom layers.

Finally, in the *inter-domain* setting, for the very lowest levels of all DNNs representing natural scenes, we expect to be able to share layers across domains. For instance, since faces *are* objects, we could expect the lowest (and computationally most expensive) layers to be sharable across face-classification and object-classification tasks. In fact, it is conceivable that *all* models share the lowermost convolutional layer(s).

`mcdnn-c` currently allows programmers to pick model schemas or prefixes of model schemas from a library appropriate for each domain 5. We currently simply use prefixes of AlexNet, VGGNet and DeepFace with input sizes matched if necessary. Given incoming model

schema s that contains such a length-n shared prefix (say $s = s_l + s_u$) with training/validation data t/v, where s_l is the maximal shared prefix and s_u the unique postfix, we assemble a trained model consisting of two parts: a pre-trained variant M_s of the s_s, and M_u, a freshly trained variant of s_u, trained on $t' = M_s(t), v' = M_s(v)$. We believe that in many models, the shared fragment M_s will be a large fraction of total models size; 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.

5 Evaluation

We have discussed several optimizations implemented in the `mcdnn-c` compiler intended to control the balance between resource usage (specifically memory and computation) and accuracy. Further, we have described a run-time system to manage these models. Although we could statically estimate exactly the decrease in resource usage from each optimization, the critical question is whether high-enough resource decreases come with low-enough accuracy loss. Further, we wish to study to what extent the promised benefits are reflected in runtime performance gains. Below, we step through each compiler optimization analyzing its impact and then discuss some measurements of the MCDNN runtime.

5.1 Datasets and evaluation procedure

Ideally, we would evaluate our techniques on mobile images in the wild. However, no large mobile-device specific dataset exists. We therefore use three large training/validation sets collected from the internet for training and validation. Given the complexity of these images we believe performance on them is likely a *lower* bound on what is possible with a wearable device looking at day-to-day life.

The Imagenet [8] dataset is the standard for object recognition, providing 1.28M/50K images for train/validation over 1000 classes. Because the VGGNet model was only released very recently, we use the AlexNet model schema, which was the standard until this year, as the basis for object recognition. The best single-model accuracy rates of Imagenet on AlexNet are roughly 51%, which were able to reproduce. Using multiple models (and voting over their outputs) adds somewhat to accuracy, but we stick to the single-model setting.

The DeepFaceNet [24] results were reported on a proprietary dataset of faces from Facebook with 4.4 million train and validation images over 4030 classes and maximum accuracy 93%. In the absence of the extremely large dataset, we collected a new dataset MSRBingFaces with 50000/5000 images over 200 celebrities from the Internet. The best accuracy we are able achieve is 79.6% which is in line with DeepFace experiments on reduced datasets. We further labeled MSRBingFaces data with the attributes race ("African American", "white", "Hispanic", "East Asian", "South Asian", "other"), age ("0-30","30-60","60+") and gender and trained classifier with each of these.

The MITPlaces205 [26] dataset was the first large scene dataset, only officially released in the past month. It contains 7.08 million images, with a best reported result of 50.0% over 205 categories, which were able to reproduce. MITPlaces205 is complemented by the Sun405 dataset [25] which provides a roughly 14340 images labeled with 102 attributes, of which we chose three ("man-made","natural light" and "no visible horizon").

Given that it takes a week to train the MITPlaces and Imagenet, and roughly a day to train on MSRBingFaces, we do not perform every experiment on every dataset. In particular, we perform all experiments with the MSRBingFaces dataset, many with Imagenet, and only sharing experiments with MITPlaces given its recent release.

5.2 Efficacy of local optimization

We applied local optimizations as per the `localOpt` algorithm to AlexNet and DeepFaceNet. The configurations produced are described in the AlexNet and DeepFaceNet rows of Table 4 (we will discus DeepFaceCompactNet later). The describes 7 optimized model schema (A1-7) for AlexNet and 11 (D1-D11) for DeepFaceNet, with the rewrites required to generate them (relative to A0 and D0) listed in the "optimizations" column. Schema A5, for instance, is the result of applying 4 rewrites.

Table 5 lists the resource consumption versus accuracy of these models. We focus on two points. First, **it is indeed possible to get very substantial reductions in resource usage with modest reductions in accuracy**. For instance A2 sacrifices roughly 8.5% accuracy (relative, and 4.1% absolute) for a 4.1× reduction in memory footprint. Similarly, A6 yielded a 3× reduction in compute use at 7.2%. In the case of DeepFace, simplifying the model does not in many cases reduce accuracy. This is likely because our smaller MSRBingFace dataset was overfit by the original DeepFaceNet model schema D0. We therefore take the simpler model D8 which has the maximum accuracy as a fairer baseline. Note that configuration D9, for instance, gets 2.3× memory use reduction *and* 1.7× for 3% relative accuracy loss. These results are reassuring, since it demonstrate that automatic optimizations can compensate for model schema that were chosen without careful consideration to resource use.

A second, and tantalizing point worth noting is that performance degrades gracefully: it does not plunge when schema are simplified dramatically. Note A3 and D10, which achieve 13.8 and 47.7(!)× improvements in memory use at just 12.5% and 15.1%. CNNs performance is therefore fairly robust to major surgery in re-

Model	Index	Optimization
AlexNet	A0	No optimization
	A1	Convert all `inner[4096]` to `[2048]`
	A2	Convert all `inner[4096]` to `[1024]`
	A3	Convert all `inner[4096]` to `[128]`
	A4	Convert `gconv2[5,256,1]` to `[5,128,1]`, all `inner[4096]` to `[2048]`
	A5	Convert `gconv1[11,96,4] to [11,48,4]`, `gconv2[5,256,1]` to `[5,128,1]`, all `inner[4096]` to `[2048]`
	A6	Convert `gconv2[5,256,1]` to `[5,256,2]`
	A7	Convert `gconv2[5,256,1]` to `[5,256,2]`, all `inner[4096]` to `[2048]`
	A8	Based on A7, add one `inner[2048]`
DeepFaceNet	D0	No optimization
	D1	Convert `inner[4096]` to `inner[2048]`
	D2	Convert `inner[4096]` to `inner[1024]`
	D3	Convert `inner[4096]` to `inner[256]`
	D4	Convert `gconv1[11,32,1]` to `[11,16,1]`
	D5	Convert `lconv3[9,16,1]` to `[9,8,1]`
	D6	Convert `gconv1[11,32,1]` to `[11,32,2]`
	D7	Convert `lconv3` to `gconv3`
	D8	Convert `lconv4` to `gconv4`, `lconv5` to `gconv5`
	D9	Convert `gconv1[11,32,1]` to `[11,16,1]`, `lconv3` to `gconv3`
	D10	Convert `gconv1[11,32,1]` to `[11,32,2]`, `lconv3` to `gconv3`
	D11	Based on D6, add one `inner[4096]`
	D12	Based on D10, add one `inner[4096]`
DeepFace CompactNet	C0	Compact DeepFace neural network
	C1	Convert `inner[2048]` to `inner[256]`
	C2	Convert `lconv3[9,16,1]` to `lconv3[9,8,1]`, `inner[2048]` to `inner[32]`
	C3	Convert `lconv3[9,16,1]` to `lconv3[9,8,3]`, `inner[2048]` to `inner[32]`
	C4	Convert `gconv1[11,32,2]` to `gconv1[11,32,4]`, `lconv3[9,16,1]` to `lconv3[9,8,3]`, `inner[2048]` to `inner[32]`

Table 4: Optimization descriptions on AlexNet, DeepFaceNet, and DeepFace CompactNet. See the definition of AlexNet and DeepFaceNet at Table 2, and DeepFace CompactNet at Figure 6.

source usage, a fact that is important for automated resource-use optimization.

Overall, these are meaningful gains both in absolute terms, since 10s of MB of memory are a relatively large gain for mobile devices and also relative to CNN size, because they will allow more models to be used simultaneously.

5.3 Global optimization

Recall that in local optimization we monotonically reduced resource consumption of operations. This naturally resulted in a steady reduction of classification accuracy in return for the performance, but kept the compilation process simple. To touch on the potential of non-monotonicity, we allowed a post-pass to local optimization where we *add functionality* via an additional single inner-product to the locally optimized models.

Models A8, D11 and D12 of Table 6, generated by augmenting models A7, D6 and D10 illustrate the results, which are striking. Note that these latter models, produced by local optimization, consume substantially less resources than their originals (A0, D0, D0), but sacrifice accuracy noticeably. **Global optimizations gain back a very substantial part of the accuracy lost by local optimization while sacrificing almost none of the gains in compute operations and relatively little of memory gains**. Note that although D12 does use 9× more memory than D10, D12 itself was using roughly 48× less memory than the original in the first place.

These results indicate that although for simplicity we designed the `mcdnn-c` compiler to monotonically reduce resource consumption in its optimizations, we have likely barely scratched the surface of possible optimization. In particular adding richer non-monotonicity in a tractable manner will likely be profitable.

5.4 Model sharing

Table 7 illustrates the potential of sharing, in this case in the *intra-domain* setting, for the face domain. The D0 Noshare row reports the result of training the DeepFace model separately on the age, gender and race data, thus generating three custom classifiers. The recognition rates are quite good, but resource consumption is that of D0. In un-shared mode, therefore if face identification were running alongside these three classifiers, we would need 4×(1GFLOP,102M floats) of processing and storage. However, row 2 ("D0 retarget") uses the result of retargeting the D0 CNN by retraining its softmax layer (as explained in Section 4.1.4). The incremental cost for the three re-targeted classifiers in this case is simply the cost of the softmax layer: e.g., the age

Index	#Flops	#Parameters	Accuracy
A0	2.54G	76.0M	49.6%
A1	2.46G	35.7M	50.0%
A2	2.40G	18.7M	45.4%
A3	2.40G	5.50M	43.7%
A4	1.80G	34.9M	43.4%
A5	1.45G	34.8M	42.2%
A6	838M	41.4M	46.0%
A7	792M	18.4M	42.7%
D0	1.00G	103.2M	75.7%
D1	974M	88.4M	76.6%
D2	959M	80.9M	74.9%
D3	948M	75.4M	74.3%
D4	605M	103.2M	77.5%
D5	933M	67.9M	76.8%
D6	203M	10.4M	69.2%
D7	1.00G	40.5M	71.2%
D8	1.00G	92.6M	78.6%
D9	605M	40.5M	76.2%
D10	203M	2.16M	66.7%

Table 5: Local optimization results

Index	#Flops	#Parameters	Accuracy
A7	792M	18.4M	42.7%
A8	**800M**	**22.6M**	**44.7%**
D6	203M	10.4M	69.2%
D11	**236M**	**27.2M**	**75.2%**
D10	202.9M	2.16M	66.7%
D12	**236M**	**18.9M**	**75.6%**

Table 6: Comparison between models with (in bold) and without global optimization for both AlexNet and DeepFaceNet

classifier requires a mere additional 24.6KFLOP/12.3K floats: **shared models can require many orders of magnitude less incremental memory and processing per additional model than without sharing**. The numbers for retargeting the scene classifier to classify other attributes are similar . Sharing by retargeting seems to enable almost unbounded scaling within a domain, possibly key to running large numbers of models efficiently both on mobile device and on the cloud.

What if the face identification model does not need to be run, but the other three do? Can we do better than incurring the (1GFLOP,102M floats) cost of the shared D0 CNN? Intuitively age, gender and race all have many fewer classes to recognize than facial identity, and a much simpler classifier would seem to suffice for them. The "D10 retarget" row confirms this intuition dramatically. This row consists of classifiers for gender, age and race that are obtained by retargeting model D10, again at very small incremental cost. But recall from Table 5 that D10 only consumes (202.9MFLOP,2.14M floats) in the first place, with roughly 2% absolute performance drop: roughly **47.6× less memory and** 4.9× **fewer FLOPs** than simply using an unoptimized baseline (D0) for retargeting.

Model	Result	Age	Gender	Race
D0 NoShare	#Flops	1.00G	1.00G	1.00G
	#Params	102M	102M	102M
	Accuracy	74.0%	94.9%	83.3%
D0 Retarget	#Flops	24.6K	16.4K	41.0K
	#Params	12.3K	8.19K	20.5K
	Accuracy	75.3%	94.7%	83.6%
D6 Retarget	#Flops	24.6K	16.4K	41.0K
	#Params	12.3K	8.19K	20.5K
	Accuracy	73.2%	93.2%	80.2%
D10 Retarget	#Flops	24.6K	16.4K	41.0K
	#Params	12.3K	8.19K	20.5K
	Accuracy	72.9%	92.7%	81.5%

Table 7: Results of retargeting the trained models to age/gender/race attribute classification. NoShare model is trained on a complete model on the attribute data. Retarget indicates that only the softmax layer is retrained on the attribute data, while bottom layers are shared from the various models trained on MSRBingFace ID.

```
input[152,152,3]
gconv1[11,32,2]+relu
mpool[3,2]
gconv2[9,16,1]+relu
lconv3[9,16,1]+relu
inner[2048]+relu
inner[200]
softmax
```

Figure 6: Definition of specialized DeepFace CompactNet (C0)

Thus sharing not only enables massive scaling, in combination with local optimization, it also scales down the minimal cost substantially. This result says that it is possible to run CNN-based age, face and race (and presumably many 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.

5.5 Model specialization

Table 8 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 MSRBing dataset) and the rest are random other faces, can you use this information to train a model with exceptionally high accuracy and low resource consumption as per Section 4.1.3? 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 MSRBingFaces-Context60 through -Context95 below.

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 8 reports an accuracy as X/Y% (e.g., 96.0/86.3%), X is

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

Table 8: Train the DeepFaceCompactNet models on context specific dataset. Context specific dataset consists of face images from in-context people and other random outside-context people. We pick 7 people randomly from MSRBingFace dataset for in-context data. Proportion of in-context images in the entire dataset varies from 60% to 95%.

Model	14 persons	21 persons
C0	95.0%	90.8%
C1	95.6%	91.1%
C2	95.4%	90.3%
C3	92.8%	86.2%
C4	86.3%	79.7%

Table 9: For 90% context specific case, we extended the number of context specific people to 14 and 21, and retargeted C0-C4.

in-context-only accuracy whereas Y is the overall accuracy.

To maximize the challenge, we picked as baseline for this study the best model we could derive from D0 for classifying the MSRBingFaces dataset. We call this model C0, and it is defined in Figure 6. We derived it by hand after extensive experimentation. Notice from the "C0" row (columns 2-4) of Table 8 that C0 has the better recognition rates than D0 and any of its automatically derived variants D1-D12 (Table 7), and it has exceptionally low resource usage. We wished to test if model specialization could significantly improve over this baseline.

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 MSRBingFaces-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 Table 9, which reports 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 from in-context. Overall classification accuracy (not reported) varies similarly as in Table 8.

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.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

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

Algorithm 3 `reduceOpt`: Reducing DNN resource use by greedy resource-usage-reducing rewrites.

function REDUCEOPT($s_0, t, v, (a_0, l_0, c_0), N, \tau, \Delta a = A, \Delta v = V$)
  ▷ **input:** Model schema s_0; training/validation data t/v; limits $a_0/l_0/c_0$ on accuracy/model size/computational operations of output schema; limit N on number of greedy steps; limit Δa on drop in accuracy allowed in a single greedy step; vector Δv of greedy step size.
  ▷ **output:** Model schema s, the cheapest schema derived from s_0 under constraints (a_0, l_0, c_0); MCDNN model M and validation accuracy a resulting from training s on t, v.

  $a, l, c, i, s, M, ps \leftarrow 1.0, l_0, c_0, 0, s_0,$ TRAIN$(s_0, t, v, \tau), \{\}$
  while $(l > l_0$ **—** $c > c_0)$ **&** $a > a_0$ **&** $i{+}{+} < N$ **do**
    $a', s', M' \leftarrow a, s, M$
    $s, p \leftarrow$ TRANSFORM$(s, i, ps, \Delta v)$
    $M \leftarrow$ TRAIN(s, t, v, τ)
    if $a' - M.a > \Delta a$ **then** ▷ Avoid plunges in accuracy
      $a, s, M, ps \leftarrow a', s', M', +\{p\}$
    else
      $ps \leftarrow \{\}$
    end if
  end while
  return M
end function

function TRANSFORM$(s, i, ps, \Delta v)$
  ▷ Find and decrement the parameter p^* in s (but not in ps) that maximally decreases storage/computation usage (if step number, i, is even/odd). Ensure that the resulting schema s^* is well-formed and return it along with p^*.

  $s^*, k^*, p^* \leftarrow m, \infty,$ **nil**
  ▷ Find parameter+schema that maximize cost reduction.
  for all p, v **in** s.paramValues() **s.t.** $p \notin ps$ **do**
    $s' \leftarrow$ REPARAMETERIZE$(m, p, v - \Delta v[p])$
    $s', c' \leftarrow$ COST(m')
    ▷ Alternately reduce storage and computational cost.
    $k' \leftarrow s'$ **if** $i\%2 = 0$ **else** c'
    $s^*, k^*, p^* \leftarrow s', k', p'$ **if** $k' < k^*$
  end for
  return s^*, p^*
end function

function TRAIN$(s, t, v, \tau =$ **nil**$, h =$ **nil**$)$
  ▷ Train schema s on dataset t/v.
  $m, a \leftarrow$ DNNTRAIN(s, t, v) ▷ Use base DNN trainer
  return $\{s : s, m : m, h : h, a : a, v : v, \tau_i : \tau.0, \tau_o : \tau.1\}$
end function

function COST(s)
  ▷ Use Table 1 to sum up the cost of all operations in model schema s. Return the total storage and computation costs.
end function

function REPARAMETERIZE(s, p, v)
  ▷ Set parameter p in model schema s to v. Prune the resulting schema by (recursively) removing any operations with no outputs. Return the resulting schema.
end function

Algorithm 4 `preSpecialize`: Specializing DNNs to partially specified training data.

function preSpecialize($Ms, t, v, hs, N = 20$)
  ▷ **input:** Model-schema-to-model-map Ms, hints hs, training/validation data t/v, number of samples N to specialize on.
  ▷ **output:** Ms with original entries and new entries derived by specializing original entries to t/v per hints hs.

  for all h **in** hs **do**
    ▷ Add most accurate specialized model
    $Ms' \leftarrow$ preSpecializeModel(M, t, v, h, N) **for** M **in** Ms
    $M^* \leftarrow \text{argmax}_{M \in Ms'}\ M.a$
    $Ms \leftarrow\!+ M^*$
  end for
  return Ms
end function

function preSpecializeModel($M, t, v, h, N = 20$)
  ▷ Specialize a single model to a single hint
  $tvs \leftarrow$ sample(t, v, h, N)
  $rs \leftarrow$ {reTarget(M, t', v', h) **for** t', v' **in** tvs}
  $(s^*, m^*, a^*), \sigma_a \leftarrow arg\ median_{s,m,a \in rs} a, \sigma_{s,m,a \in rs} a$
  $M' \leftarrow \{s : s^*, m : m^*, h : h, a : a^*, \sigma_a : \sigma_a, v : v, \tau : M.\tau\}$
  return M'
end function

function sample($t, v, h = \{n, p, \ldots\}, N$)
  ▷ Sample N train/validation sets from t, v as per hint h: p percent of each sampled set must come from one of n classes in t, with the rest chosen randomly from remaining classes.
  $tvs \leftarrow \{\}$
  for i **in** $0 \ldots N-1$ **do**
    $c \leftarrow$ set of n fresh random class labels from t
    ▷ r_s denotes the subset of r with labels in s; $r \otimes m$ draws m samples uniformly w.o. replacement from r.
    $t' \leftarrow t_c \cup (t_{\bar{c}} \otimes (|t_c|(\frac{1}{p} - 1)))$
    $v' \leftarrow v_c \cup (v_{\bar{c}} \otimes (|v_c|(\frac{1}{p} - 1)))$
    $tvs \leftarrow\!+ (t', v')$
  end for
  return tvs
end function

function reTarget($M, t, v, h = \{n, p, \ldots\}$)
  ▷ Retrain top layer of $M.m$ on t, v
  $s, m, \tau = M.s, M.m, M.\tau$
  $s_b, m_b \leftarrow s[:-1], m[:-1]$ ▷ Extract all but top layer of s
  $s_t \leftarrow$ *[input(s[-1]), inner({nelts:n}), softmax()]*
  $M_t \leftarrow$ train$(s_t, m_b(t), m_b(v), \tau)$
  $s' = s_b + M_t.s[1:]$ ▷ Add top layer (minus input layer)
  $m' = m_b + M_t.m[1:]$
  return $s', m', M_t.a$
end function

Algorithm 5 `trainSharedLayers`: Identify and incorporate layers shared across DNNs.

function trainSharedLayers($Ms, t, v, l =$ "mcdnnlib")
  ▷ **input:** Models Ms; training data t, v used for these models; library of models l.
  ▷ **output:** List of models formed by sharing prefixes of models in Ms with models in l of matching type.
  for all M, M_l **in** $Ms \times l.Ms$ **s.t.** $M \prec M_l$ **do**
    if $n =$ nSharedLayers$(M, M_l) > 0$ **& !**$M_l.h$ **then**
      $M_l, s_u \leftarrow M_l[:n], input(M.s[n-1]) + M.s[n:]$
      $M_u \leftarrow$ train$(s_u, M_l(t), M_l(v))$
      $M' \leftarrow M_l + M_u$ ▷ Compose shared model
      if $M.h$ **then** ▷ Specialize shared model if needed
        $M' \leftarrow$ preSpecializeModel$(M', t, v, M.h)$
      end if
      $Ms \leftarrow\!+ M'$
    end if
  end for
  return Ms
end function

function nSharedLayers(M,M')¡
  ▷ Length of common prefix of the schemas of M and M'
  $s, s', n, r \leftarrow M.s, M'.s, \text{MIN}(|s|, |s'|), n$ ▷ Seq. assignment
  for i **in** $0 \ldots n-1$ **do**
    if $s[i] \neq s'[i]$ **then**
      $r \leftarrow i$**; break**
    end if
  end for
  return r
end function