System for Serving Deep Neural Networks Efficiently

Haichen Shen

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Arvind Krishnamurthy, Chair

Matthai Philipose, Chair

Thomas E. Anderson

Xi Wang

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science and Engineering

University of Washington

Abstract

System for Serving Deep Neural Networks Efficiently

Haichen Shen

Co-Chairs of the Supervisory Committee:
Professor Arvind Krishnamurthy
Computer Science & Engineering

Affiliate Professor Matthai Philipose
Computer Science & Engineering

Today, Deep Neural Networks (DNNs) can recognize faces, detect objects, and transcribe speech, with (almost) human performance. DNN-based applications have become an important workload for both edge and cloud computing. However, it is challenging to design a DNN serving system that satisfies various constraints such as latency, energy, and cost, and still achieve high efficiency. First, though server-class accelerators provide significant computing power, it is hard to achieve high efficiency and utilization due to limits on batching induced by the latency constraint. Second, resource management is a challenging problem for mobile-cloud applications because DNN inference strains device battery capacities and cloud cost budgets. Third, model optimization allows systems to trade off accuracy for lower computation demand. Yet it introduces a model selection problem regarding which optimized model to use and when to use it.

This dissertation provides techniques to improve the throughput and reduce the cost and energy consumption significantly while meeting all sorts of constraints via better scheduling and resource management algorithms in the serving system. We present the design, implementation, and evaluation of three systems: (a) Nexus, a serving system on a cluster of accelerators in the cloud that includes a batch-aware scheduler and a query analyzer for complex query; (b) MCDNN, an approximation-based execution framework across mobile devices and the cloud that manages resource budgets

proportionally to their frequency of use and systematically trades off accuracy for lower resource use; (c) Sequential specialization that generates and exploits low-cost and high-accuracy models at runtime once it detects temporal locality in the streaming applications. Our evaluation on realistic workloads shows that these systems can achieve significant improvements on cost, accuracy, and utilization.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This dissertation wouldn't be possible without help from many people. First I would like to thank my advisors Arvind Krishnamurthy and Matthai Philipose. They provide much guidance and advice over the years and help me grow into an independent researcher. Arvind is insightful on how to formulate research problems and identify the essence of problems, and always encouraged me to dive deeper. Matthai worked closely with me in many projects and has been helpful whenever I encountered problems. I am also grateful to have David Wetherall, Aruna Balasubramanian, and Anthony LaMarca advise me in early years during my Ph.D. Further, I would like to thank other committee members of my dissertation, Tom Anderson and Xi Wang. Their feedback and advice are invaluable.

I would also like to thank many collaborators and colleagues in my Ph.D. life. Seungyeop Han worked with me on several projects and he is an exemplary researcher for me. I am also honored to work with Tianqi Chen on TVM. He is very talented and inspired me from many perspectives. I thank Yuchen Jin and Bingyu Kong for their help on Nexus. I thank all my labmates and friends, Qiao, Danyang, Ming, Sophia, Raymond, Will, Nacho, Jialin, Naveen, Antoine, Irene, Dan, Adriana, Taesoo, Colin, Shumo, Ravi, and Qifan. I learned a lot through discussion and working closely with them. I thank Melody Kadenko, Elise Degoede, and Lindsay Michimoto for their excellent supports in administrative things that make the way to my Ph.D. degree without any complications.

Finally, I want to thank my family. My wife Yuhan has been supportive throughout my entire Ph.D. I am deeply grateful that she delivered our lovely daughter Hannah during my last year of Ph.D. My parents taught me numerous principles and lessons that made me become who I am now. I thank their unconditional support and love.

Dedication

To my wife Yuhan and my daughter Hannah

Chapter 1

Introduction

Deep Neural Networks (DNNs) have become the dominant approach to solve a variety of computing problems such as handwriting recognition [80], speech recognition [83], and machine translation [119]. In many computer vision tasks such as face [123] and object [59] recognition, deep learning can almost achieve human-level performance. Sophisticated DNN-based applications can be built by composing several DNNs together. For example, an intelligent assistant can incorporate a speech recognition DNN and a text summarization DNN to transcribe the conversation and summarize it as a memo. A traffic monitoring application can use an object detector first to find all cars in the video frame, and then apply a DNN classifier to recognize car make and model as well as extract license plate information.

Although DNNs have been shown to provide excellent performance, they are also known to be computationally intensive. Unlike traditional machine learning algorithms, DNNs require significant amount of computational resources during inference, not just during training. DNN inference routinely consumes hundreds of MB of memory and GFLOPs of computing power [77, 123, 59, 121]. Further, DNN-based applications usually need to execute multiple DNN models to process a single user request. Also, we expect these applications to invoke DNN models frequently, as they usually process high-datarate streams such as video and speech.

In recent years, specialized hardware and accelerators such as GPUs [2], FPGAs [29], and ASICs like TPU [71] have become more common both in the data center as well as on edge devices. Today, modern mobile phones come with mobile GPUs [6, 12] and even specialized chips such as neural engines [116] in iPhone A11/A12 chips and Huawei's NPU [34]. Though these accelerators lower cost and energy consumption for DNN inference compared to CPU, it is still non-trivial to serve DNN

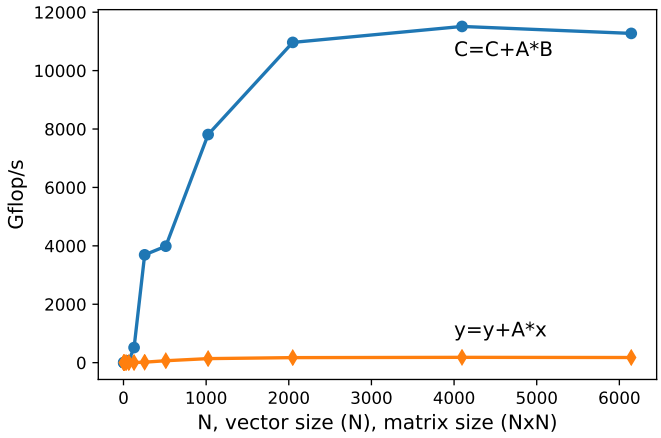


Figure 1.1: The impact of batching. GPU utilization for GEMM vs GEMV operations. Data measured on the NVIDIA Titan X (Pascal) GPU. Matrix-matrix multiplication can be viewed as a > 60× more efficient batched form of matrix-vector multiplication.

workloads at peak efficiency.

First, while server-class accelerators provide very high capacity (with the peak FLOPS ratings for modern GPUs exceeding 10 TFLOPS), it is necessary to group inputs into a batch before passing them through the computation in order to reap this enormous computing power. For example, Figure 1.1 compares sustained FLOPS by highly optimized implementations of matrix-vector multiplication ("GEMV") versus matrix-matrix multiplication ("GEMM"), which can be viewed as a batched GEMV operation. In relative terms, GEMM can sustain more than 60× the utilization of GEMV. However, batching increases the latency for each request since all inputs are computed in lockstep and has to be used with care in the context of applications requiring latency-sensitive inferences. This introduces **resource allocation and scheduling problem** for cloud operators regarding how to distribute DNN workloads onto a cluster of accelerators at high utilization while satisfying the latency requirements.

Second, in the context of edge execution, DNNs strain device batteries even when the devices are equipped with mobile GPUs or custom DNN ASICs. For example, a mobile GPU can provide 290 GOP/s at 10 W, indicating that each operation consumes 34 pJ. Consider a DNN model with 5 billion floating operations per invocation that is invoked 15 times per second. Even a large 3000 mAh mobile phone battery will be drained within less than 5 hours by merely this single model. Further, limited memory capacity could also become a bottleneck for large DNN models or when there is more than

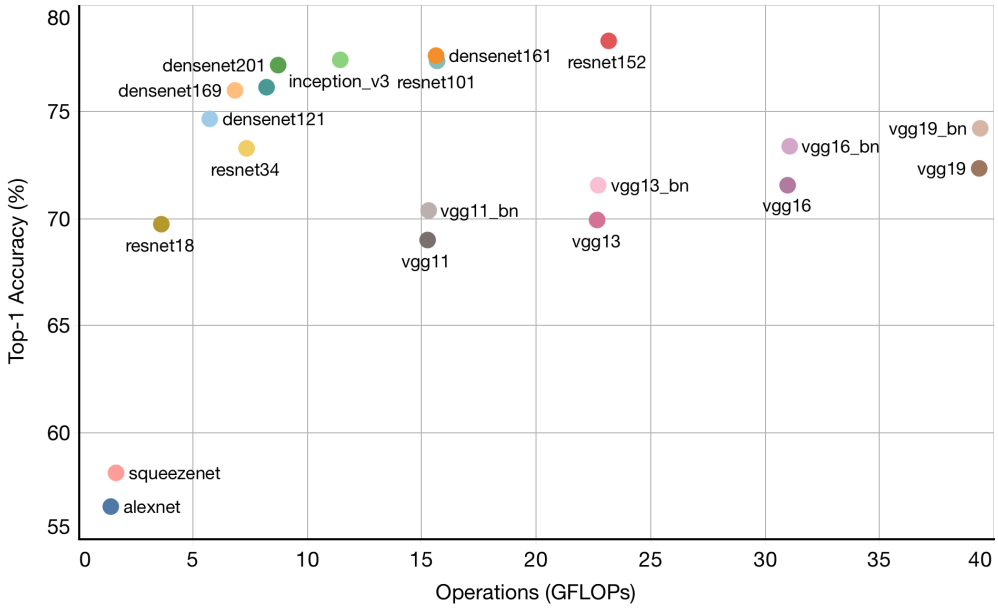


Figure 1.2: Top-1 accuracy on ImageNet object classification versus number of operations for various models.

one application performing DNN operations. Nonetheless, the cost of cloud computation is not cheap either. If we target at $10 per client per year, we can get no more than 10 GFLOP/s from EC2 instances. The disparity between the high computation demands of DNNs and the limited resources on mobile devices and cost budgets in cloud leads to a challenging **resource management problem** — serving system needs to carefully decide where to execute DNNs, mobile device or cloud, before exhausting resource budgets.

Third, machine learning community has been proposing new model architectures with higher accuracy or lower computation demand. For example, Figure 1.2 depicts the accuracy and amount of operations of 20 different models for object classification tasks. In addition, another line of work focuses on *model optimization* techniques that can reduce memory [139, 138, 26, 55, 65] and processing demands [68, 74, 111] to any DNN, typically at the expense of some loss in classification accuracy. These techniques help generate a model catalog with a range of accuracy and computation costs for a given task. This provides an opportunity for systems to trade off accuracy and resource use by tackling the **model selection problem**, i.e., how to achieve the optimal accuracy and performance within the resource budgets and constraints at runtime. Further, given the relevance of the streaming setting in

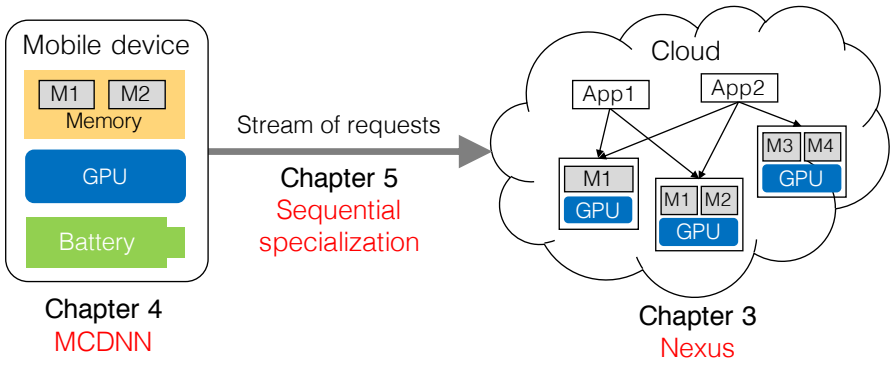


Figure 1.3: System suite overview

DNN workload, it is also possible to generate optimized models on the fly that are specialized to the current temporal context for better accuracy and performance. Hence, the model selection problem has to take into account not only the resource settings but also the inference contexts, and a DNN serving system has to continually generate optimized models and decide whether it is effective to use the specialized models.

The key question this dissertation addresses is how to improve the efficiency — throughput, utilization, and accuracy — of DNN inference while meeting various constraints such as latency, cost budget, energy consumption, and memory capacity. We argue that in order to achieve high efficiency, it is necessary to design a serving system that (a) is aware of the properties of devices, accelerators, and models, including energy budget, memory capacity, and model performance curve with regard to different batch sizes and operators; (b) understands characteristics of DNN models such as their accuracy, energy costs, and execution latency; and (c) tracks the statistics of the workload such as incoming request rates and input distribution.

In this thesis, I built a system suite that optimizes the execution of DNN inference, depicted in Figure 1.3. It contains three major components.

- Nexus is designed for DNN workload orchestration on a cluster of accelerators in the cloud. It includes techniques for performing resource allocation and scheduling multi-tenant DNN workloads. It improves GPU efficiency and utilization while meeting latency requirements.

- MCDNN considers execution on cloud-backed mobile devices. It manages the available resource budgets for computation, memory, energy, and cloud cost, and it systematically determines which optimized models to use and where to execute them to improve the overall accuracy.

- Sequential specialization is an optimization for streaming applications. It exploits the temporal locality that a small set of objects or persons will be frequently found within a short time period in the context of streams. This optimization reduces the computational cost and execution latency while providing comparable or increased accuracy.

1.1 Nexus

As mentioned above, specialized hardware accelerators for DNNs have emerged in the recent past. Accelerators are specialized to process DNNs orders of magnitude faster and cheaper than CPUs in many cases. However, cost-savings from using them depends critically on operating them at sustained high utilization. In this work, we primarily focus on GPUs, but the same principles can be also applied to other accelerators.

Conceptually, the problem can be thought of sharding inputs coming through a distributed frontend onto DNNs served at "backend" GPUs. Several interacting factors complicate this viewpoint. First, given the size of GPUs, it is often necessary to place different types of networks on the same GPU. It is then important to select and schedule them so as to maximize their combined throughput while satisfying latency bounds. Second, many applications consist of *groups* of DNNs that feed into each other. It is important to be able to specify these groups and to schedule the execution of the entire group on the cluster so as to maximize performance. Third, as mentioned above, dense linear algebra computations such as DNNs execute much more efficiently when their inputs are *batched* together. Batching fundamentally complicates scheduling and routing because (a) it benefits from cross-tenant and cross-request coordination and (b) it forces the underlying bin-packing-based scheduling algorithms to incorporate batch size. Fourth, the increasingly common use of *transfer learning* [41, 137] in today's workloads has led to specialization of networks, where two tasks that

formerly used identical networks now use networks that are only approximately identical. Since batching only works when multiple inputs are applied to the *same* model in conventional DNN execution systems, the benefits of batching are lost.

Nexus is a cloud serving system for DNN execution on GPU cluster that addresses these problems to attain high execution throughput under latency Service Level Objectives (SLOs). It uses three main techniques to do so. First, it relies on a novel *batching-aware scheduler* (Section 3.4.1) that performs bin packing when the balls (i.e., DNN tasks) being packed into bins (i.e., GPUs) have variable size, depending on the size of the batch they are in. This schedule specifies the GPUs needed, the distribution of DNNs across them and the order of their execution so as to maximize execution throughput while staying within latency bounds. Second, it allows groups of related DNN invocations to be written as *queries* and provides automated query optimization to assign optimal batch sizes to the components of the query so as to maximize the overall execution throughput of the query while staying within its latency bounds. Finally, Nexus breaks from orthodoxy and allows batching of *parts of networks* with different batch sizes. This enables the batched execution of specialized networks.

1.2 MCDNN

Given the relevance of such applications to the mobile setting and the emergence of powerful mobile GPUs [6, 12] and ASICs [116, 34], there is a strong case for executing DNNs on mobile devices. We present a framework, called MCDNN, for executing multiple applications that use large DNNs on (intermittently) cloud-connected mobile devices to process streams of data such as video and speech. We target high-end mobile-GPU-accelerated devices, but our techniques are useful broadly.

We anticipate that in the near future, multiple applications will seek to run multiple DNNs on incoming sensor streams such as video, audio, depth and thermal video. The large number of simultaneous DNNs in operation and the high frequency of their use will strain available resources, even when optimized models are used. We use two insights to address this problem. First, model optimization typically allows a graceful trade-off between accuracy and resource use. Thus, a system could adapt to high workloads by using less accurate variants of optimized models. Second, we adopt two powerful model optimizations for the streaming and multi-model settings: (a) using

sequential specialization to produce light-weight models with low cost and high accuracy, (b) sharing computation of queries from multiple models on the same stream that may have semantic similarity.

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

1.3 Sequential Specialization

It's quite common that DNN applications receive streaming workloads. Consider the workload of recognizing entities such as objects, people, scenes and activities in every frame of video footage of day-to-day life. In principle, these entities could be drawn from thousands of classes: many of us encounter hundreds to thousands of distinct people, objects, scenes and activities through our life. Over short intervals such as minutes, however, we tend to encounter a very small subset of classes of entities. For instance, a wearable camera may see the same set of objects from our desk at work for an hour. We characterize and exploit such *short-term class skew* to significantly reduce the latency of classifying video using DNNs.

We measure a diverse set of videos and demonstrate that for many recognition tasks, day-to-day video often exhibits significant short-term skews in class distribution. For instance, that in over 90% of 1-minute windows, at least 90% of objects interacted with by humans belong to a set of 25 or fewer objects. The underlying recognizer, on the other hand, can recognize up to 1000 objects. We show that similar skews hold for faces and scenes in videos.

We then demonstrate that when the class distribution is highly skewed, *"specialized" DNNs* trained to classify inputs from this distribution can be much more compact than the *oracle DNN*,

the unoptimized classifier. For instance, we present a DNN that executes 200× fewer FLOPs than VGGFace [100] model, but has comparable accuracy when over 50% of faces come from the same 10 or fewer people. We present similar order-of-magnitude faster specialized CNNs for object and scene recognition.

The key challenges for specialization to work at inference time are (a) how to produce accurate versions of DNNs specialized for particular skews within a short time, (b) how to balance exploration (i.e., using the expensive oracle to estimate the skew) with exploitation (i.e., using a model specialized to the current best available estimate of the skew). We first devise an optimization that only retrains the top layers of models, and thus reduce the training time to a few seconds. Second, we formalize the "bandit"-style sequential decision-making problem as the Oracle Bandit Problem and propose a new exploration/exploitation-based algorithm we dub Windowed ε-Greedy (WEG) to address it.

1.4 Contributions

This dissertation demonstrates that we can improve the throughput and reduce the cost and energy consumption significantly while meeting the relevant constraints (e.g., latency, cost, energy). I explore the design space that ranges from a common optimization for streaming applications that can achieve both high accuracy and low cost, to two end-to-end system designs that can achieve higher throughput and accuracy for two deployment scenarios. I summarize the contributions made in this dissertation as follows.

- design and implementation of a cloud serving system for a cluster of accelerators that is aware of batching characteristics when scheduling both standalone DNN tasks as well as composite or complex DNN jobs;

- a thorough analysis of optimized models regarding the trade-off between accuracy and energy consumption, execution latency, and cost;

- design and implementation of an approximation-based execution framework that systematically chooses optimized models and schedules execution between mobile devices and the

cloud to satisfy resource constraints as well as optimize the overall accuracy;

- demonstrate that specialized models can be much more compact and still achieve high accuracy when the class distribution is highly skewed;

- devise the Windowed ε-Greedy algorithm to address the challenge in balancing the exploration that uses an oracle to estimate the underlying distribution associated with incoming streams, and exploitation that uses a specialized model to suit the current distribution.

1.5 Organization

The remaining chapters are organized as follows.

Chapter 2 presents the background of basic operators in deep neural networks and several classic neural network architectures. We survey various model optimization techniques that can reduce the memory and computation demands of models. This chapter also describes a few DNN application scenarios and surveys related work.

Chapter 3 addresses the problem of serving DNNs at sustained high utilization from a cluster of GPUs in a multi-tenant cloud setting. We present the system design of Nexus and batch-aware cluster-scale resource management that performs careful scheduling of GPUs.

Chapter 4 considers the DNN inferences on mobile devices that operate in concert with the cloud. We present the design of the MCDNN system that systematically trades off accuracy for resource use and balances the DNN execution between mobile and cloud to satisfy the resource constraints and cost budget.

Chapter 5 presents an optimization for streaming applications that trains "specialized" models on the fly using time context information to speed-up DNN inference. We describe the mechanism of how to detect the context changes and when to toggle "specialized" models on and off.

Finally, Chapter 6 concludes the dissertation and discusses future work.

Chapter 2

Background

This chapter presents a brief introduction to deep neural networks including the operators used in DNNs and several common model architectures. We then provide a survey of model optimization techniques. Finally, we discuss related work of three systems in this dissertation.

2.1 Deep Neural Networks

Deep neural networks (DNNs) are networks of dense linear algebra operations. Typical inputs to the network (or "model") include images to be classified, audio snippets to be transcribed, or text to be understood and the outputs are the corresponding answers. Inputs and outputs are typically encoded as dense vectors of floats.

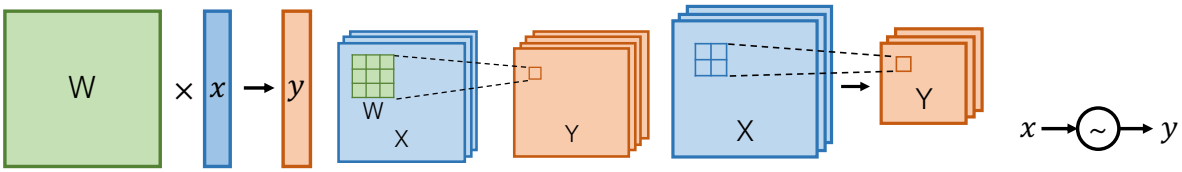
Each operator (or "layer") in the model is one of a small set of linear algebra operations [1], depicted in Figure 2.1 :

matrix-vector multiplication ("GEMV") (also called fully-connected layer), multiplies input vector x by a *weight matrix* W of size, e.g., 1024×1024,

$$y = W \cdot x \tag{2.1}$$

tensor[2] convolution, convolves input in 3-D matrix by a set of convolution kernels W of size $K \times K$,

[1] Researchers have proposed other operators, but this set is representative.



(a) Fully-connected (b) Convolution (c) Pooling (d) Activation

Figure 2.1: Four common operators in deep neural networks.

typically $K = 3, 5$, or 7:

$$y_{mij} = \sum_{p,q \le K, n \le N} W_{mnyx} x_{n(i+p)(j+q)}, \tag{2.2}$$

pooling, replaces each element with its local maximum (or average) within a $K \times K$ window ($K = 2$ or 3) , and usually shifts in stride $S = 2$ or 3:

$$y_{cij} = \max_{p,q \le K} \left(x_{c(i \cdot S+p)(j \cdot S+q)} \right) \tag{2.3}$$

activation, replaces each element with its simple non-linear transform, e.g., ReLU (Rectified Linear Units), sigmoid, tanh, etc. σ:

$$y_{ij} = \sigma(x_{ij}). \tag{2.4}$$

The matrix-vector multiplication and convolution layers are parameterized by weight arrays that are learned from training data. The network description is called a *model architecture*, and the trained network with weights instantiated is a *model*. Convolutional Neural Networks (CNNs), mostly used in vision tasks, tends to have linear feed-forward structure. For example, [77, 117] but also include directed acyclic graphs (DAGs) [121, 122, 59]. Meanwhile, Recurrent Neural Networks (RNNs), heavily used in natural language processing, have the recurrent graph structures and are fully unrolled over incoming data before execution.

We now explore a few CNN model architectures that are used in the later chapters. These model

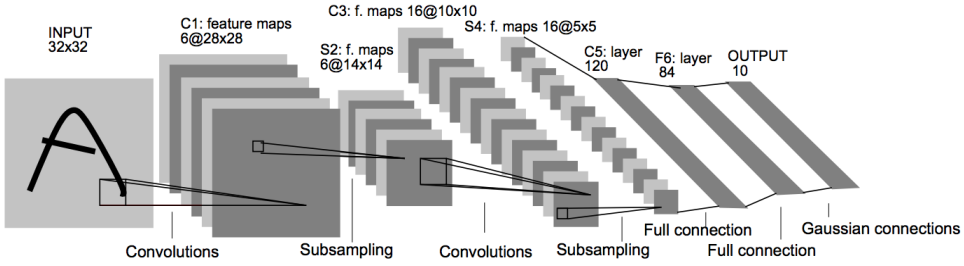


Figure 2.2: LeNet [80] model architecture.

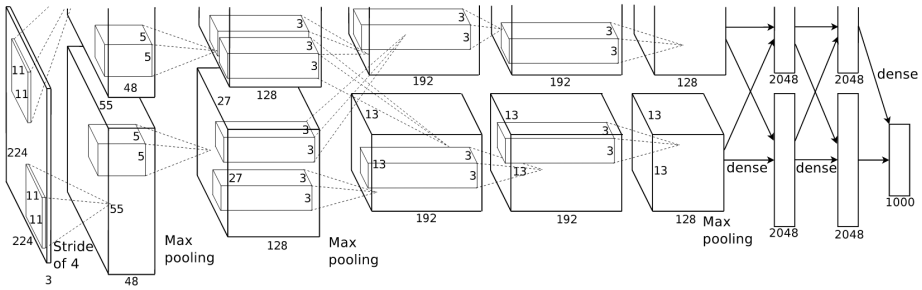


Figure 2.3: AlexNet [77] model architecture.

architectures are introduced in chronological order. First, Figure 2.2 shows the architecture of LeNet [80], designed by Yann LeCun et al. in 1998. LeNet is used to recognize handwritten digits. It consists of two convolution layers, two pooling layers, and two fully-connected layers. The model is fairly small compared to the rest of models due to the computing power limit at the time. Its total computation cost is 4.6 million FLOPs and contains 0.4 million parameters to be learned.

Alexnet [77], depicted in Figure 2.3, is proposed in 2012 by Alex Krizhevsky et al., and is targeted for object recognition instead of digit recognition. It consists of 5 convolution layers, 2 max pooling layers, and 3 fully-connected layers. This model requires 1.4 billion FLOPs of computation and 61 MB of memory, and achieves 56.6% accuracy over 1000 different object classes.

VGG-Net [117] (Figure 2.4) extends AlexNet design by adding more convolution layers and using smaller kernel sizes. It has four versions with 11, 13, 16, and 19 layers (counting convolution and fully-connected layers). Models with more layers achieve higher accuracy but require more computation (39 GFLOPs in 19 layers) and memory (144 MB in 19 layers), as shown in Figure 1.2.

Continuing the trend of deeper structure, ResNet [59] introduces the skip connection from

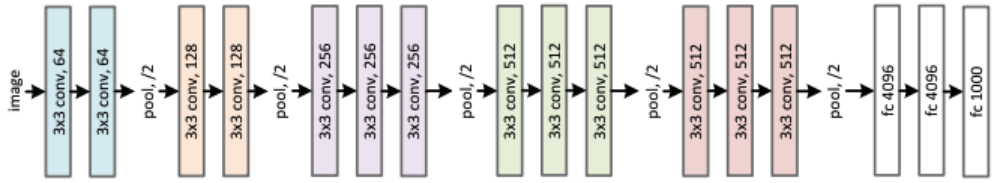


Figure 2.4: VGG [117] model architecture (16-layer version in this figure).

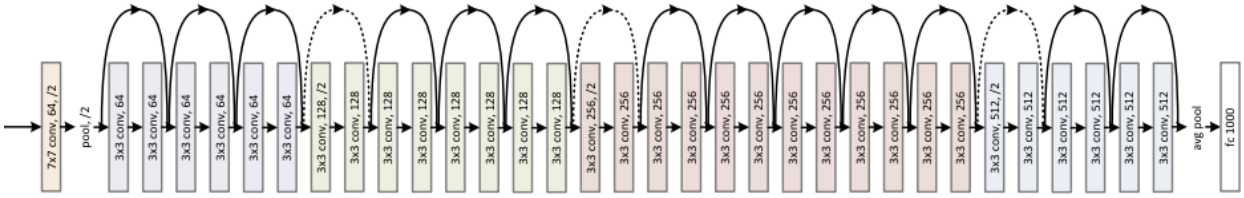


Figure 2.5: ResNet [59] model architecture (34-layer version in this figure).

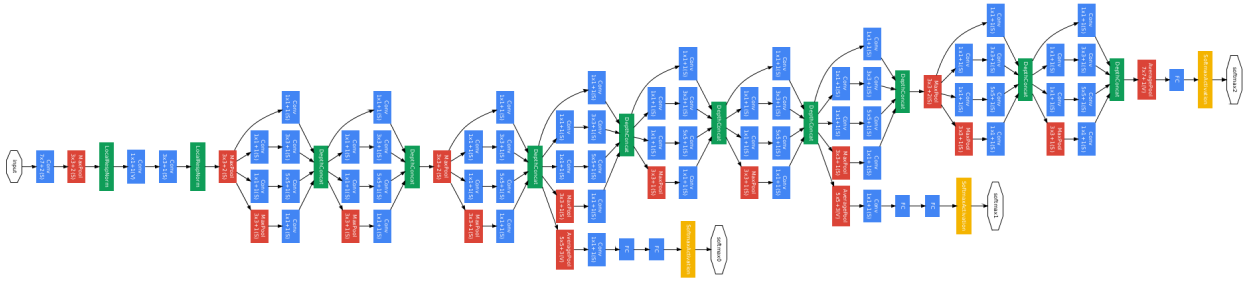


Figure 2.6: Inception [121] model architecture.

previous layers, as shown in Figure 2.5. This skip connection structure solves the vanishing gradient problem [18, 47] and therefore is able to achieve higher accuracy. The number of layers in ResNet ranges from 18 to 152 layers and can achieve up to 78.3% accuracy.

Inception [121, 122], on the other hand, reduces the amount of computation by using smaller convolution kernel sizes and parallel architecture, depicted in Figure 2.6. Inception requires 11.5 GFLOPs of computation and 27 MB of memory while achieving 77.5% accuracy.

Deep neural networks go through two phases. During the *training* phase, the weights and kernels (generically, "parameters") W are estimated based on a set of desired input-output pairs (x, y), typically using some variant of the gradient descent algorithm. This phase often takes hours to days and many recent systems [35, 28, 86] work has focused on increasing the speed of (distributed) gradient descent.

Once the network is trained, the parameters W are fixed, and the network N can be used for *inference*: for a given input x (e.g., an image), N is *evaluated* to compute output (e.g., a list of windows in the image with the name of the person in that window) $y = N(x)$. Evaluating a typical network may require executing several GFLOPs so that for many applications, inference cost is the bulk of the cost of execution.

2.2 Model Optimization

Among the four operators described in Section 2.1, matrix-vector products and convolutions typically consume well over 95% of execution time. As a result, recent *model optimization* techniques for reducing resource use in DNNs have targeted these operators in three main ways:

1. *Matrix/tensor factorization* replaces the weight matrices and convolution kernels by their low-rank approximations [138, 68, 111, 75]. Replacing a size-$M \times M$ weight matrix $W_{M\times M}$ with its singular value decomposition $U_{M\times k}V_{k\times M}$ reduces storage overhead from M^2 to $2Mk$ and computational overhead from M^3 to $2M^2k$. This can be further extended to factorize higher dimension tensors for convolution [75]. The most recent results [75] have reported reductions of 5.5× in memory use and 2.7× in FLOPs at a loss of 1.7% accuracy for the "AlexNet" model we use for scene recognition and 1.2×, 4.9× and 0.5% for the "VGG16" model we use for object recognition.

2. *Matrix pruning* [26, 55] sparsifies matrices by zeroing very small values, use low-bitwidth representations for remaining non-zero values and use compressed (e.g., Huffman coded) representations for these values. The most recent results [55] report 11/8× reduction in model size and 3/5× reduction on FLOPs for AlexNet/ VGGNet, while sacrificing essentially no accuracy. However, for instance, sacrificing 2% points of accuracy can improve memory size reduction to 20×.

3. *Quantization* replaces floating-point values by fixed-point values [145] or, to an extreme, binaries [108]. These works can reduce the memory footage by > 30× and number of operations

by 50× at an accuracy loss of around 10%.

4. *Architectural changes* explore the space of model architectures, including varying the number of layers, size of weight matrices including kernels, etc. For instance, VGGNet [117] reduces the number of layers from 19 to 11 results in a drop in accuracy of 4.1% points. Researchers [146] also propose systematical search over the model architectures for targeted accuracy and computation demands, though this approach requires amounts of resources and time to complete.

2.3 Related Work

Previous sections give a brief introduction to deep neural networks and model optimization techniques. In this section, we discuss previous work that is related to the three systems in the dissertation.

2.3.1 Nexus

The projects most closely related to Nexus are Clipper [31] and Tensorflow Serving [17]. Clipper is a "prediction serving system" that serves a variety of machine learning models including DNNs, on CPUs and GPUs. Given a request to serve a machine learning task, Clipper selects the type of model to serve it, batches requests, and forwards the batched requests to a backend container. By batching requests, and adapting batch sizes online under a latency SLO, Clipper takes a significant step toward Nexus's goal of maximizing serving throughput under latency constraints. Clipper also provides approximation and caching services, complementary to Nexus's focus on executing all requests exactly but efficiently. Tensorflow Serving can be viewed as a variant of Clipper that does not provide approximation and caching, but also has additional machinery for versioning models.

To the basic batched-execution architecture of Clipper, Nexus builds along the dimensions of *scale*, *expressivity*, and *granularity*. These techniques address the challenges and thus reflect Nexus's focus on executing DNNs on GPUs at high efficiency and scale.

Scale: Nexus provides the machinery to scale serving to large, changing workloads. In particular, it automates the allocation of resources (including GPUs) and the placement and scheduling of

models across allocated resources. It provides a distributed frontend that scales to large numbers of requests, with a work-sharding technique that plays well with batching requests on the back end. These functions are performed on a continuing basis to adapt to workloads, with re-allocation and re-packing structured to cause minimum churn.

Expressivity: Nexus provides a query mechanism that (a) allows related DNN execution tasks to be specified jointly, and (b) allows the user to specify the latency SLO just at the whole-query level. Nexus then analyzes the query and allocates latency bounds and batch sizes to constituent DNN tasks so as to maximize the throughput of the whole query.

Granularity: Where Clipper limits the granularity of batched execution to whole models, Nexus automatically identifies common *subgraphs* of models and executes them in a batch. This is critical for batching on specialized models, which often share all but the output layer, as described previously. Pretzel [81, 82] also tackles the model granularity problem by matching duplicate sub-modules in the database. But they only focus on the benefit of memory preservation and speed-up in model loading but don't shed light on execution efficiency improvement of the common parts.

Recent work also explores DNN serving systems for application specific workload. Sirius [58] and DjiNN [57] discuss the designs of warehouse scale computers to efficiently serve voice and vision personal assistant workload, a suite of 7 DNN models. They mainly analyze the performance and total cost of ownership of server designs from the architecture perspective. VideoStorm [142] explores the resource-quality trade-off for video analytics with multi-dimensional configurations such as resolution and sampling rate. While VideoStorm improves performance on quality and lag, the solution is not generally applicable to other applications.

Serving DNNs at scale is similar to other large-scale short-task serving problems. These systems have distributed front ends that dispatch low-latency tasks to queues on the backend servers. Sparrow [99] focuses on dispatch strategies to reduce the delays associated with queuing in such systems. Slicer [11] provides a fast, fault-tolerant service for dividing the back end into shards and load balancing across them. Both systems assume that the backend server allocation and task placement is performed at a higher (application) level, using cluster resource managers such as Mesos [60] or Omega [114]. Nexus shares the philosophy of these systems of having a fast data plane that dispatches

incoming messages from the frontend to backend GPUs and a slower control plane that performs more heavyweight scheduling tasks, such as resource allocation, packing and load balancing. Also, the Nexus global scheduler communicates with cluster resource managers to obtain or release resources.

Much work in the machine learning and systems community has focused on producing faster models. For instance, many model optimization techniques [108, 10, 139] produce faster versions of models, often at small losses in accuracy. Given a variety of models of varying accuracy, they can be combined to maintain high accuracy and performance, using static or dynamic techniques [115, 72, 62, 53, 31]. Nexus views the optimization, selection and combination models as best done by the application, and provides no special support for these functions. On the other hand, once the models to be executed are selected, Nexus will execute them in a fast, efficient manner. Systems such as [35, 86, 133] makes training more scalable via parameter servers. Optimus [102] improves the resource efficiency of a training cluster by using online resource-performance models to predict model convergence time. Gandiva [132] provides time slicing and job migration of training jobs in order to improve GPU utilization. However, Gandiva targets at a more relaxed problem compared to Nexus because it doesn't have latency constraints for training jobs and there is no variation in the incoming inputs each time as training data is available from the beginning.

Beyond machine learning systems, there are considerable related materials from database and big data community regarding query optimization [21, 69] and distributed data analytic systems [36, 67, 140, 97]. Yet most of these systems operate at a coarse time granularity, usually minutes and hours instead of sub-second job latency in Nexus. Streaming query processing systems [8, 7, 141, 96] have a closer setting to Nexus, but have no latency guarantee. Jockey [44] provides latency SLO guarantee by allocating resources based on precomputed statistics and trace from previous execution. It targets for parallel execution in CPUs instead of accelerators and doesn't reason about batching effect. Besides, unlike SQL-like query processing where jobs can start on any server, DNN tasks require to first load model into main memory before processing. However, model loading time is significantly higher than its execution time. Therefore, Nexus plans ahead and pre-loads model before execution starts whereas Jockey decides dynamically.

2.3.2 MCDNN

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

Recent work in the machine learning community has focused on reducing the overhead of DNN execution. For important models such as speech and object recognition, research teams have spent considerable effort in producing manually optimized versions of individual DNNs that are efficient at run-time [121, 111, 83]. Several recent efforts in the machine learning community have introduced automated techniques to optimize DNNs, mostly variants of matrix factorization and sparsification to reduce space [139, 138, 26, 54, 108] and computational demands [68, 111, 75, 108]. Many of these efforts support the folk wisdom that DNN accuracy can broadly be traded off for resource usage. MCDNN is complementary to these efforts, in that it is agnostic to the particular optimizations used to produce model variants that trade off execution accuracy for resources. Our primary goal is to develop a novel approximating runtime that selects between these variants while obeying various practical resource constraints over the short and long term. In doing so, we further provide both a more comprehensive set of measurements of accuracy-resource trade-offs for DNNs than any we are aware of, and devise two novel DNN optimizations that apply in the streaming and multi-programming settings.

The hardware community has made rapid progress in developing custom application-specific integrated circuits (ASICs) to support DNNs [24, 88, 27]. We view these efforts as complementary to MCDNN, which can be viewed as a compiler and runtime framework that will benefit most underlying hardware. In particular, we believe that compiler-based automated optimization of DNNs, cross-application sharing of runtime functionality, and approximation-aware stream scheduling will all likely still be relevant and useful whenever (even very efficient) ASICs try to support multi-application, continuous streaming workloads. In the long term, however, it is conceivable that multiple DNNs could be run concurrently and at extremely low power on common client hardware,

making system support less important.

Recent work in the mobile systems community has recognized that moving sensor-processing from the application (or library) level to middleware can help avoid application-level replication [98, 113, 87]. MCDNN may be viewed as an instance of such middleware that is specifically focused on managing DNN-based computation by using new and existing DNN optimizations to derive approximation versus resource-use trade-offs for DNNs, and in providing a scheduler that reasons deeply about these trade-offs. We have recently come to know of work on JouleGuard, that provides operating-system support, based on ideas from reinforcement learning control theory, for trading off energy efficiency for accuracy *with guarantees* across approximate applications that provide an accuracy/resource-use "knob" [61]. MCDNN is restricted to characterizing (and developing) such "knobs" for DNNs processing streaming workloads in particular. On the other hand, MCDNN schedules to satisfy (on a best-effort basis, with no guarantees) memory use and cloud-dollar constraints in addition to energy. MCDNN's technical approach derives from the online algorithm community [20, 16]. Understanding better how the two approaches relate is certainly of strong future interest.

Off-loading from mobile device to cloud has long been an option to handle heavyweight computations [50, 32, 105]. Although MCDNN supports both off- and on-loading, its focus is on *approximating* a *specific class* of computations (stream processing using DNNs), and on deciding automatically where to best execute them. MCDNN is useful even if model execution is completely on-client. Although we do not currently support split execution of models, criteria used by existing work to determine automatically whether and where to partition computations would be relevant to MCDNN.

Cross-application sharing, has similarities to standard common sub-expression elimination (CSE) [30] or memoization [51, 91], in that two computations are partially unified to share a set of prefix computations at runtime. Sharing is also similar to "multi-output" DNNs trained by the machine learning community, where related classification tasks share model prefixes and are jointly trained. MCDNN's innovation may be seen as supporting multi-output DNNs in a *modular* fashion: different from jointly-trained multi-output models, the "library" model is defined and trained separately from the new model being developed. Only the upper fragment of the new model is trained subsequently. Further, to account for the fact that the (typically expensive) library model may not be available for

sharing at runtime, the MCDNN compiler must train versions that do not assume the presence of a library model, and the MCDNN runtime must use the appropriate variant.

2.3.3 Sequential Specialization

There is a long line of work on cost-sensitive classification, the epitome of which is perhaps the cascaded classification work of Viola and Jones [127]. The essence of this line of work [136, 134] is to treat classification as a sequential process that may exit early if it is confident in its inference, typically by learning sequences that have low cost in expectation over training data. Recent work [84] has even proposed cascading CNNs as we do. All these techniques assume that testing data is i.i.d. (i.e., not sequential), that all training happens before any testing, and rely on skews in *training* data to capture cost structure. As such, they are not equipped to exploit short-term class skews in test data.

Traditional sequential models such as probabilistic models [130, 38, 103] and Recurrent Neural Networks (RNNs) [40, 73] are aimed at classifying instances that are not independent of each other. Given labeled sequences as training data, these techniques learn more accurate classifiers than those that treat sequence elements as independent. However, to our knowledge, none of these approaches produces classifiers that yield *less expensive* classification in response to favorable inputs, as we do.

Similar to adaptive cascading, online learning methods [126, 56, 78] customize models at test time. For training, they use labeled data from a sequential stream that typically contains both labeled and unlabeled data. As with adaptive cascading, the test-time cost of incrementally training the model in these systems needs to be low. A fundamental difference in our work is that we make no assumption that our input stream is partly labeled. Instead, we assume the availability of a large, resource-hungry model that we seek to "compress" into a resource-light cascade stage.

Estimating distributions in sequential data and exploiting it is the focus of the multi-armed bandit (MAB) community [13, 79]. The Oracle Bandit Problem (OBP) we define differs from the classic MAB setting in that in MAB the set of arms over which exploration and exploitation happen are the same, whereas in OBP only the oracle "arm" allows exploration whereas specialized models allow exploitation. Capturing the connection between these arms is the heart of the OBP formulation. Our Windowed ε-Greedy algorithm is strongly informed by the use of windows in [46] to handle

non-stationarities and the well-known [120] ε-greedy scheme to balance exploration and exploitation.

Finally, much recent work has focused on reducing the resource consumption of (convolutional) neural networks [54, 108, 14, 55]. These techniques are oblivious to test-time data skew and are complementary to specialization. We expect that even more pared-down versions of these optimized models will provide good accuracy when specialized at test-time.

Chapter 3

Nexus: Scalable and Efficient DNN Execution on GPU Clusters

This chapter describes a cloud serving system called Nexus that targets to serve Deep Neural Networks (DNNs) efficiently from a cluster of GPUs. In order to realize the promise of very low-cost processing made by accelerators such as GPUs, it is essential to run them at sustained high utilization. Doing so requires cluster-scale resource management that performs detailed scheduling of GPUs, reasoning about groups of DNN invocations that need to be co-scheduled, and moving from the conventional whole-DNN execution model to executing fragments of DNNs. Nexus is a fully implemented system that includes these innovations. On large-scale case studies on 16 GPUs, Nexus shows 1.8-41× better throughput than state-of-the-art systems while staying within latency constraints > 99% of the time.

3.1 Background

Nexus provides *efficient* and *timely* cloud-based acceleration of applications that analyze video. Figure 3.1 provides some examples of the applications we target. For instance, a city may have hundreds of traffic cameras, a game streaming site [3] may index tens of thousands of concurrent streams for discovery, a media company may index dozens of live broadcasts, and a home monitoring company may analyze camera feeds from millions of customers. Nexus is designed to support many such users and applications simultaneously, pooling workloads across them to achieve efficiency. Below, we examine the structure of these applications and outline the challenges and opportunities in serving them at scale.

(a) Traffic monitoring (b) Game stream discovery (c) News indexing

(d) Building access (e) Home monitoring

Figure 3.1: Example video streams. Windows to be analyzed by Nexus using DNNs are marked with boxes.

3.1.1 Vision pipelines and DNN-based analysis

At the highest level, a vision-based application aggregates visual information from one or more video streams using custom "business" logic. Each stream is processed using a pipeline similar to that in Figure 3.2. CPU-based code, either on the edge or in the cloud, selects *frames* from the stream for processing, applies business logic to identify what parts (or *windows*) of the image need deeper analysis, applies a set of Deep Neural Networks (DNNs) (a *query*) to these windows, and aggregates the results in an application specific way, often writing to a database. A query may represent a single DNN applied to the window, but often it may represent a sequence of dependent DNN applications, e.g., running an object detector on the window and running a car make/model detector on all sub-windows determined to be cars.

Typically, a stream is sampled a few times a second or minute, and the DNN query should complete execution in tens to hundreds of milliseconds (for "live" applications) or within several hours for ("batch" applications). The execution of DNNs dominates the computation pipeline, and

[1]Per-device prices for 1000 invocations assuming peak execution rates on on-demand instances of AWS c5.large (Intel AVX 512), p2.xlarge (NVIDIA K80), p3.2xlarge (NVIDIA V100) and GCP Cloud TPU on 9/14/2018.

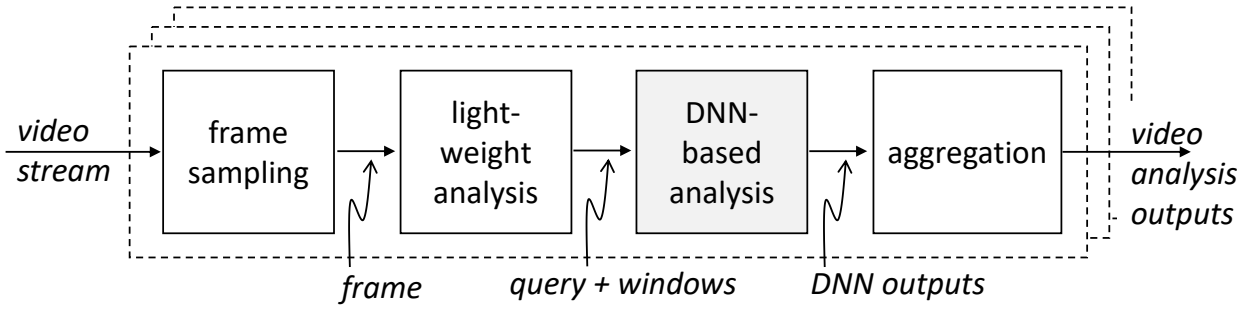


Figure 3.2: A typical vision processing pipeline. Nexus is designed to provide DNN-based analysis for tens of thousands of streams.

Model	CPU lat.	GPU lat.	CPU cost (0.1TF peak)	TPU cost (180TF peak)	GPU cost (125TF peak)
Lenet5	6ms	<0.1ms	$0.01	$0.00	$0.00
VGG7	44	<1	0.13	0.01	0.00
Resnet50	1130	6.2	4.22	0.48	0.12
Inception4	2110	7.0	8.09	0.93	0.23
Darknet53	7210	26.3	24.74	2.85	0.70

Table 3.1: DNN execution latencies and estimated costs per 1000 invocations [1]. Acceleration may be necessary to meet latency deadlines, but can also be cheaper, given low cost/TFLOPS (written TF above).

the cost of executing them dominates the cost of the vision service. Nexus provides a standalone service that implements the DNN-based analysis stage for vision pipelines.

3.1.2 Accelerators and the challenge of utilizing them

As Table 3.1 shows, a key to minimizing the cost of executing DNNs is the use of specialized accelerators such as GPUs and TPUs (Tensor Processing Units, essentially specialized ASICS), which are highly optimized to execute the dense linear algebra computations that comprise DNN models. The table shows the execution latency and the dollar cost of 1000 invocations for a few common models on CPUs and GPUs. Execution times on CPUs can be orders of magnitude slower than that on GPUs. For many applications, therefore, latency constraints alone may dictate GPU-accelerated execution.

Perhaps more fundamentally, GPUs and TPUs promise much lower cost per operation than even highly accelerated CPUs: Table 3.1 lower-bounds the cost of executing a model by assuming that

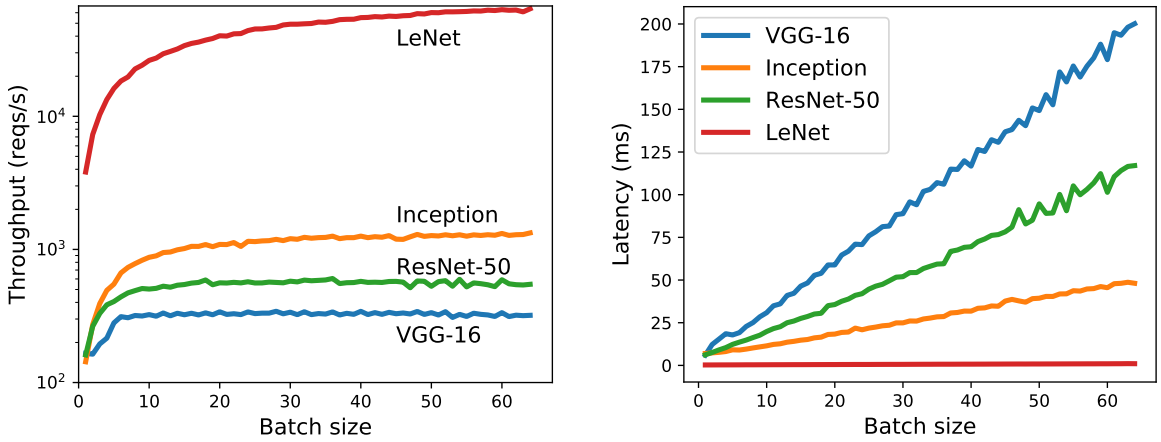


Figure 3.3: Impact of batching DNN inputs on model execution throughput and latency on an NVIDIA GTX1080.

models can be executed at peak speed on each platform. Even compared to state of the art CPUs, accelerators can yield a cost advantage of up to 9× (for TPUs) and 34× (for GPUs). On the other hand, accelerators have extremely high computational capacity (e.g., 125 TFLOPS for the NVIDIA V100). *To realize their cost savings, it is critical to sustain high utilization of this capacity*. Sustaining high utilization is hard, however. For instance, the LeNet model of Table 3.1 consumes 20 MOPs to run, implying that *a single* V100 would require 125 TFLOPS ÷ 20 MOPs = 6.25M inputs/second to run at full utilization!

No single stream, or even most applications, can yield such rates. By aggregating inputs across streams, Nexus is designed to funnel adequate work to each accelerator. However, as we discuss next, having "enough" work is not sufficient to achieve high utilization: it is important to group the right type of work in the right place.

3.1.3 Placing, packing and batching DNNs

DNNs are networks of dense linear algebra operations (e.g., matrix multiplication and convolution), called *layers* or *kernels*. Networks are also called *models*. By default, the GPU simply executes the kernels presented to it in the order received. The kernels themselves are often computationally

intensive, requiring MFLOPs to GFLOPs to execute, and range in size from one MB to hundreds of MBs. These facts have important implications for GPU utilization.

First, loading models into memory can cost hundreds of milliseconds to seconds. When serving DNNs at high volume, therefore, it is usually essential to *place* the DNN on a particular GPU by pre-loading it on to GPU memory and then re-using it across many subsequent invocations. Placement brings with it the traditional problems of efficient *packing*. Which models should be co-located on each GPU, and how should they be scheduled to minimize mutual interference?

Second, it is well known that processor utilization achieved by kernels depends critically upon *batching*, i.e., grouping input matrices into higher-dimensional ones before applying custom "batched" implementations of the kernels. Intuitively, batching allows kernels to avoid stalling on memory accesses by operating on each loaded input many more times than without batching. As Figure 3.3 shows, batching can improve the throughput of model execution significantly, e.g., improvements of 2.5-24× for batch sizes of 8-64 for the VGG and LeNet models relative to batch size 1, with intermediate gains for other models. Although batching increases execution latency, it usually stays within application-level bounds.

Although batching is critical for utilization, it complicates many parts of the system design. Perhaps most fundamentally, the algorithm for packing models on GPUs needs to change because the resource quantum used on each input is "squishy", i.e., it varies with the size of the batch within which that input executes. Further, the latency of execution also depends on the batch size. This new version of bin packing, which we dub *squishy bin packing*, needs to reason explicitly about batching (Section 3.4.1). Batching also complicates query processing. If a certain latency SLO (Service Level Objective) is allocated to the query as a whole, the system needs to partition the latency across the DNN invocations that comprise the query so that each latency split allows efficient batched execution of the related DNN invocation (Section 3.4.2). We call this *complex query scheduling*.

Finally, batching is conventionally only feasible when the same model is invoked with different inputs. For instance, we expect many applications to use the same well-known, generally applicable, models (e.g., Resnet50 for object recognition). However, the generality of these models comes at the price of higher resource use. It has become common practice [94, 48] to use smaller models

specialized (using "transfer learning") to the few objects, faces, etc. relevant to an application by altering ("re-training") just the output layers of the models. Since such customization destroys the uniformity required by conventional batching, making specialized models play well with batching is often critical to efficiency.

3.2 Examples

In this section, we use two simple examples to explain the optimization problems associated with scheduling DNN workloads from Section 3.1.3. The first example explores squishy bin packing, and the second, scheduling complex queries.

3.2.1 Squishy bin packing

Consider a workload that consists of three different types of jobs that invoke different DNN models. Let the desired latency SLOs for jobs invoking models A, B, and C be 200ms, 250ms, and 250ms, respectively. Table 3.2 provides the batch execution latency and throughput at different batch sizes (i.e., the "batching profile") for each model.

We first explore the most basic scenario where all three types of jobs are associated with high request rates so that multiple GPUs are required to handle each job type. To maximize GPU efficiency, we need to choose the largest possible batch size while still meeting the latency SLO. We note that the batch execution cost for a given job type cannot exceed half of the job's latency SLO; a request that missed being scheduled with a batch would be executed as part of the next batch, and its latency would be twice the batch execution cost. For example, the latency SLO of job 1 has 200 ms, so the maximum batch size that job 1 can use is 16, according to Table 3.2. Therefore, the maximum throughput that job 1 can achieve on a single GPU is 160 reqs/sec, and the number of GPUs to be allocated for job 1 should $r_1/160$, where r_1 is the request rate of job 1. Similarly, the number of GPUs for job 2 and 3 should be $r_2/128$ and $r_3/128$, where r_2 and r_3 are the request rates for jobs 2 and 3 respectively. Figure 3.4a depicts the desired schedules for the three types of jobs.

We next consider a situation where the request rates for the jobs are not high and each job requires only a fraction of a GPU's computing power. In this case, the scheduler needs to consolidate multiple

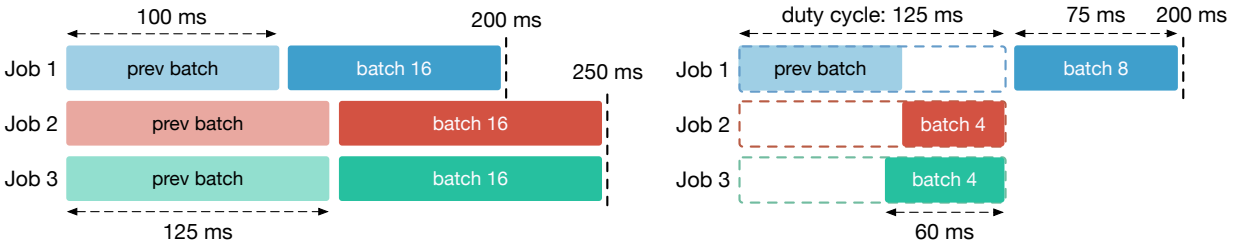
Model	Batch size	Batch lat. (ms)	Throughput (reqs/s)
A	4	50	80
	8	75	107
	16	100	160
B	4	50	80
	8	90	89
	16	125	128
C	4	60	66.7
	8	95	84
	16	125	128

Table 3.2: Batching profiles for models used in the example. Batch Lat is the latency for processing a batch.

types of DNN tasks onto the same GPU to optimize resource utilization. Consider a workload where job 1 receives 64 reqs/sec, and jobs 2 and 3 receive 32 reqs/sec. We consider schedules wherein one or more model types are assigned to each GPU. A GPU then executes batches of different types of jobs in a round robin manner, and it cycles through the different model types over a time period that we refer to as the *duty cycle*. The worst case latency for a job is no longer twice the batch execution cost but rather the sum of the *duty cycle* and the batch execution cost for that job type.

Given this setup, we observe that we can schedule model A in batches of 8 as part of a duty cycle of 125ms; the resulting throughput is the desired rate of 64 reqs/sec, the batch execution cost for 8 tasks is 75ms, and the worst case execution delay of 200ms matches the latency SLO (see Figure 3.4b). We then check whether the GPU has sufficient slack to accommodate jobs associated with models B or C. Within a duty cycle of 125ms, we would need to execute 4 tasks of either B or C to meet the desired rate of 32 reqs/sec. The batch execution cost of 4 model B tasks is 50ms, which can fit into the residual slack in the duty cycle. On the other hand, a batch of 4 model C tasks would incur 60ms and cannot be scheduled inside the duty cycle. Further, the worst case latency for model B is the sum of the duty cycle and its own batch execution cycle, 175ms$(= 125 + 50)$, which is lower than its latency SLO 250ms. Thus, it is possible to co-locate models A and B on the same GPU, but not model C.

We now make a few observations regarding the scenario discussed above and why the associated optimization problem cannot be addressed directly by known scheduling algorithms. First, unlike



(a) Saturated workload

(b) Residual workload

Figure 3.4: Resource allocation example.

Model	Batch Latency (ms)	Throughput (reqs/s)
X	40	200
	50	250
	60	300
Y	40	300
	50	400
	60	500

Table 3.3: This table shows the batch execution latency and throughput of model X and Y used in the example query.

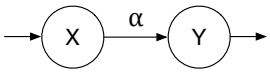


Figure 3.5: Query pipeline

vanilla bin-packing that would pack fixed-size balls into bins, here the tasks incur lower costs when multiples tasks of the same type are squished together into a GPU. Second, in addition to the capacity constraints associated with the computing and/or memory capabilities associated with a GPU, there are also latency constraints in generating a valid schedule. Third, there are many degrees of freedom in generating a valid schedule. The batch sizes associated with different model executions is not only a function of the request rate but also of the duty cycle in which the batch is embedded. In Section 3.4.1, we describe how to extend traditional algorithms to accommodate these extra considerations.

Latency budget		Avg Throughput (reqs/s)		
X	Y	α = 0.1	α = 1	α = 10
40	60	192.3	142.9	**40.0**
50	50	235.3	**153.8**	34.5
60	40	**272.7**	150.0	27.3

Table 3.4: This table shows the average throughput with three latency split plans for varying α.

3.2.2 Complex query scheduling

In the previous example, we considered simple applications that use only one model, but, in practice, applications usually comprise of dependent computations of multiple DNN models. For example, a common pattern is a detection and recognition pipeline that first detects certain objects from the image and then recognizes each object. The developer will specify a latency SLO for the entire query, but since the system would host and execute the constituent models on different nodes, it would have to automatically derive latency SLOs for the invoked models and derive schedules that meet these latency SLOs. We discussed the latter issue in the previous example, and we now focus on the former issue.

Consider a basic query that executes model X and feeds the output of X to model Y, as shown in Figure 3.5. Suppose we have a 100ms latency budget for processing this query, and supposed that every invocation of X yields α outputs (on average). When $\alpha < 1$, model X operates as a filter; when $\alpha = 1$, model X basically maps an input to an output; when $\alpha > 1$, model X yields multiple outputs from an input (e.g., detection of multiple objects within a frame).

Assume that Table 3.3 depicts the batch execution latency and throughput of models X and Y. The system has to decide what latency SLOs it has to enforce on each of the two types of models such that the overall latency is within 100ms and the GPU utilization of the query as a whole is maximized. For the purpose of this example, we consider a limited set of latency split plans for models X and Y: (a) 40ms and 60ms, (b) 50ms and 50ms, (c) 60ms and 40ms. It would appear that plan (a) should work best since the sum of the throughputs is largest among the three plans, but a closer examination reveals some interesting details.

For workloads involving a large number of requests, let us assume that p and q GPUs execute

model X and Y, respectively. We then have $\alpha \cdot p \cdot T_X = q \cdot T_Y$, where T_X and T_Y are throughputs of models X and Y, such that the pipeline won't be bottlenecked by any model. We define the average throughput as the pipeline throughput divided by the total number of GPUs, which is $p \cdot T_X/(p+q)$. We evaluate the average throughputs for the three latency split plans with $\alpha = 0.1, 1, 10$. Table 3.4 shows that each of the three split plans achieves the best performance for different α values. In fact, there is no universal best split: it depends on α, which can vary over time.

We learn two lessons from this example. First latency split for complex query makes an impact on overall efficiency, and it is necessary to understand model batch performance and workload statistics to make the best decision. Second, latency split should not be static but rather adapted over time in accordance with the latest workload distribution. Section 3.4.2 describes how analysis tool in Nexus automatically and continually derives latency splits for complex queries.

3.3 System Overview

Figure 3.6 provides an overview of Nexus. Nexus works on three planes. The *management* plane allows developers to ingest and deploy applications and models, at a timescale of hours to weeks. The *control* plane, via the *global scheduler*, is responsible for resource allocation and scheduling at a typical timescale of seconds to minutes. The *data* plane, comprised of in-application *Nexus library* instances and backend modules (together, the *Nexus runtime*), dispatches and executes user requests at the timescale of milliseconds to seconds. The global scheduler interacts with the underlying cluster resource manager (e.g., Mesos [60], Azure Scale Sets [95]) to acquire CPUs/GPUs for the frontend/backend. A load balancer (not shown) from the underlying cluster spreads user requests across Nexus's distributed frontend. We sketch the three planes.

Management plane: Developers may ingest models and *applications* to Nexus. Models are stored in a *model database* and may be accompanied by either a sample data set or a batching profile (see Table 3.2). Nexus uses the sample dataset, if available, to derive a batching profile. Otherwise, the profile is updated at runtime based on user requests. Applications are containers whose entry points invoke Nexus's `App` API (Figure 3.7), which will be discussed later. Developers store application

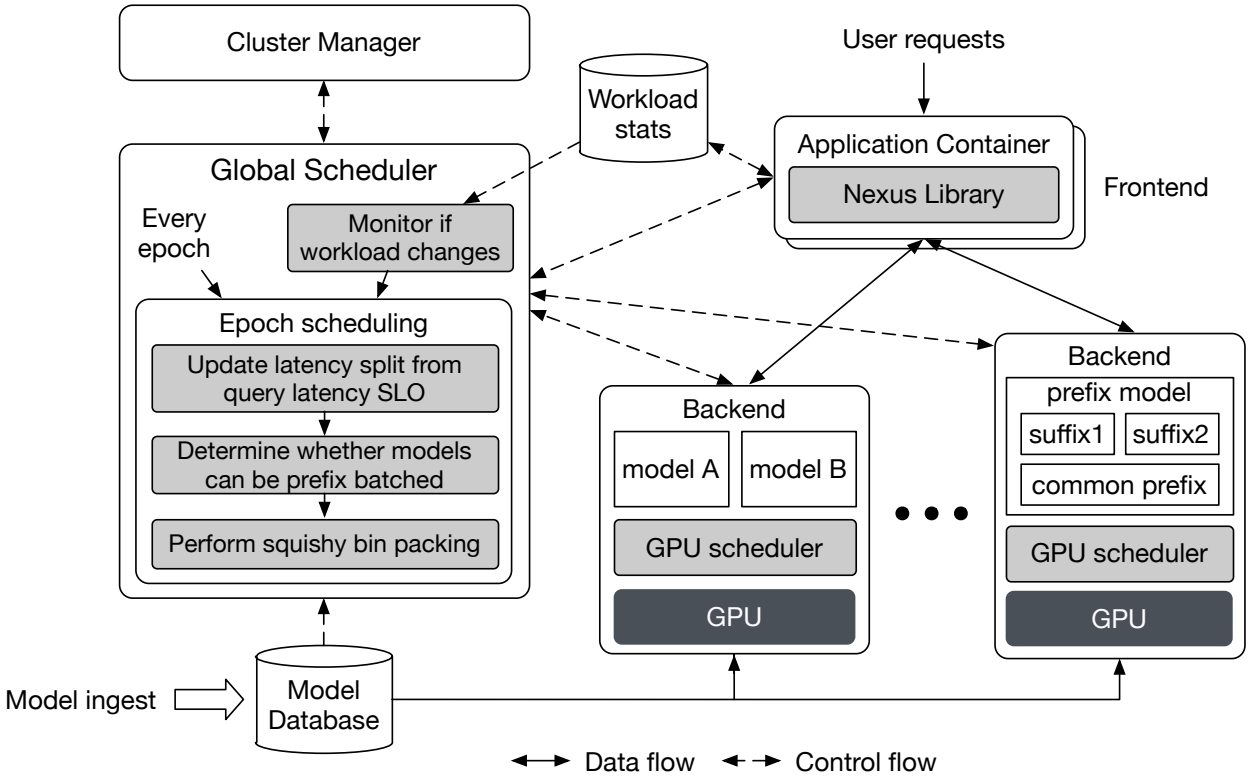


Figure 3.6: Nexus runtime system overview.

containers in cluster-provided container repositories and may instruct Nexus to ingest a container, at which point it is loaded from the repository onto a frontend CPU.

Control plane: The global scheduler is a cluster-wide resource manager that uses load and usage statistics from the runtime. It uses this profile to add or remove frontend and backend nodes from the cluster, invokes the *epoch scheduler* to decide which models to execute and at what batch size, and which backend to place the models on so as to balance load and maximize utilization. Multiple models may be mapped onto a single backend, albeit with an execution schedule that ensures they do not interfere as in Section 3.2.1. The mapping from models to backends where they are to be executed is captured in a *routing table* that is broadcast to frontends. The matching execution schedule for each backend is captured in a *schedule* that is communicated to backends. On receiving a routing table, frontends update their current routing table. On receiving a schedule, backends load appropriate models into GPU memory and set their execution schedule.

```
class ModelHandler:
  # Async RPC, executes the model on remote backends,
  # and returns an output handler immediately
  def Execute(self, input_data, output_fields)

class App:
  # Sends load model request to global scheduler,
  # and returns a ModelHandler
  def GetModelHandler(self, framework, model_name, version)
  # Implemented by developer
  def Setup(self)
  # Implemented by developer
  def Process(self, request)
```

Figure 3.7: Nexus application API.

Allocation, scheduling, and routing updates happen at the granularity of an *epoch*, typically 30-60s, although a new epoch can also be triggered by large changes in workload. Epoch scheduling involves the following:

- Produce an updated split of the latency SLO based on α values estimated from the latest profile (see Section 3.4.2).
- Combine two or more models that share a prefix and latency SLO into a new *prefix-batched* model.
- Perform profile-guided squishy bin packing to allocate the GPU resources for each model. (Section 3.4.1 has details).

Data plane: When a user request comes into (a replica of) an application container, the application invokes DNNs via the `Execute` method from the Nexus Library. The library consults the local routing table to find a suitable backend for that model, dispatches the request to the backend, and delivers responses back to the application. The application is responsible for packaging and delivering the end-result of the query to the user. A backend module uses multiple threads to queue requests from various frontends, selects and executes models on these inputs in batched mode according to the current schedule, and sends back the results to frontends. It can utilize one or more GPUs on a given node, with each GPU managed by a *GPU scheduler* that schedules tasks on it.

```
class GameApp(App):
  def Setup(self):
    self.lenet_model = self.GetModelHandler("caffe2", "lenet", 1)
    self.obj_model = self.GetModelHandler("tensorflow", "resnet", 1)
  def Process(self, request):
    ret1 = self.lenet_model.Execute(request.image, ["digit"])
    ret2 = self.obj_model.Execute(request.image, ["name"])
    return Reply(request.user_id, [ret1["name"], ret2["name"]])
```

Figure 3.8: Application example of live game streaming analysis.

3.3.1 Nexus API

Figure 3.7 defines the programming interface of Nexus. `App` class is the base class to be extended for an application. It contains an API called `GetModelHandler`, which sends a load model request to the global scheduler with framework, model name, and model version specified by developers. After global scheduler confirms that model is loaded at some backends, it then returns a `ModelHandler` object. `ModelHandler` provides only one API, `Execute`, given the input data and desired output fields. `Execute` function sends the request to one of the backends that load this model. Inside it performs load balancing according to the routing table. `Execute` function returns an output future immediately without waiting for the reply from backends. The program will be blocked and forced to synchronize only when it tries to access the data in the output future. This allows multiple requests to be executed in remote backends in parallel when they don't have dependencies between each other.

There are two functions in the `App` class that need to be implemented by developers: `Setup` and `Process`. Developers specify models to be loaded in the `Setup` function. `Process`, otherwise, contains the processing logic for the application. `Setup` is invoked once when the application is launched, whereas `Process` is invoked every time when it receives a new user request.

Figure 3.8 shows a simple application example that analyzes each frame from live game streaming. It uses two models, LeNet [80] to recognize digits and ResNet [59] to recognize icons and objects in the frame. Note that these two models use different frameworks. Thanks to Nexus API abstraction, developers can easily adopt models from different frameworks without worrying about model

```
class TrafficApp(App):
  def Setup(self):
    self.detection = self.GetModelHandler("tensorflow", "ssd_mobilenet", 1)
    self.car_rec = self.GetModelHandler("caffe2", "googlenet_car", 1)
    self.face_rec = self.GetModelHandler("caffe2", "vgg_face", 1)
  def Process(self, request):
    persons, car_models = [], []
    for rec in self.detection.Execute(request.image):
      if rec["class"] == "person":
        persons.append(self.face_rec.Execute(request.image[rec["rect"]]))
      elif rec["class"] == "car":
        cars.append(self.car_rec.Execute(request.image[rec["rect"]]))
    return Reply(request.user_id, persons, cars)
```

Figure 3.9: Traffic monitoring application example.

transformation. In the `Process` function, because there is no dependency between `ret1` and `ret2`, the request to LeNet and ResNet will execute in parallel. The program won't reply to the client until both return values are received.

Figure 3.9 presents a more complicated application, which includes a complex query. It aims to recognize the make and model of vehicles and identification of pedestrians in each frame from traffic cameras. It first invokes SSD [89] to detect objects and pedestrians from images. For every detected person and car, it then invokes face recognition model and car recognition model respectively. The number of requests sent to face and car recognition model is variable, depending on the input image.

These two examples demonstrate that it's very easy to develop and deploy an application on Nexus. The APIs are quite simple yet expressive enough to write all sorts of applications. More importantly, the complexity of DNN execution, routing, load balancing, and scaling are hidden behind these APIs and managed by Nexus system. Therefore, developers don't need to take care of those issues.

3.4 Global Scheduler

We now describe the algorithms used by the global scheduler as part of its epoch scheduling. The goal of the global scheduler is to minimize the number of GPUs allocated to processing the workload without violating latency SLOs. This problem is NP-hard (see proof at Appendix A). We devise the algorithm by extending the classic bin packing greedy strategy. We present the algorithms in three

Notation	Description
M_k	DNN model k
S_i	Session i
L_i	Latency constraint for session S_i
R_i	Request rate for session S_i
$\ell_k(b)$	Execution cost for M_k and batch size b

Table 3.5: Notation for GPU scheduling problem.

steps. First, we consider the case of scheduling streams of individual DNN task requests, given their expected arrival rates and latency SLOs (akin to the example discussed in Section 3.2.1. We then consider how to schedule streams of more complex queries/jobs that invoke multiple DNN tasks (similar to the example discussed in Section 3.2.2).

3.4.1 Scheduling streams of individual DNN tasks

We now consider scheduling streams of individual DNN tasks and build upon the discussion we presented in Section 3.2.1. The scheduler identifies for each cluster node the models hosted by the node would serve and the target serving throughputs for those models such that the schedule is computationally feasible and does not violate the latency SLOs. As discussed earlier, the scheduling problem has the structure of the bin-packing problem [19], but the solution is harder as different batching thresholds incur different per-task costs and different co-locations of model types on a node result in different worst-case model execution latencies. The "squishiness" of tasks and the need to meet latency SLOs are the considerations that we address in the algorithm presented below.

Inputs and Notation: The scheduling algorithm is provided with the following inputs.

- It is provided the request rate of invocations for a given model at a given latency SLO. We refer to the requests for a given model and latency SLO as a *session*. Note that a *session* would correspond to classification requests from different users and possibly different applications that invoke the model at a given latency constraint. Table 3.5 describes the notation used below. Formally, a session S_i specifies a model M_{k_i} and a latency SLO L_i, and there is a request rate R_i associated with it.

- The algorithm is also provided with the execution costs of different models at varying batch sizes. The latency of executing a set of M_K invocation of size b is referred to as $\ell_k(b)$. We assume that throughput is non-decreasing with regard to batch size b.

Scheduling Overview: The scheduler allocates one or more sessions to each GPU and specifies their corresponding target batch sizes. Each GPU node n is then expected to cycle through the sessions allocated to it, execute invocations of each model in batched mode, and complete one entire cycle of batched executions within a *duty cycle* of d_n. For sessions that have a sufficient number of user requests, one or more GPU nodes are allocated to a single session to execute model invocations. The algorithm described below computes the residual workload for such sessions after allocating an integral number of GPUs and then attempts to perform bin packing with the remaining sessions. For the bin packing process, the scheduler inspects each session in isolation and computes the largest batch size and the corresponding duty cycle for the executing GPU in order to meet the throughput and SLO needs. The intuition behind choosing the largest batch size is to have an initial schedule wherein the GPU operates at the highest efficiency. This initial schedule, however, isn't cost effective as it assumes that each GPU is running just one session within its duty cycle, so the algorithm then attempts to merge multiple sessions within a GPU's duty cycle. In doing so, it should not violate the latency SLOs, so we require the merging process to only reduce the duty cycle of the combined allocation. The algorithm considers sessions in decreasing order of associated work and merges them into existing duty cycles that have the highest allocations, thus following the design principle behind the best-fit decreasing technique for traditional bin packing.

Details: We now describe the global scheduling algorithm for a given workload (which is also depicted in Algorithm 3.1).

Consider a session S_i that uses model M_{k_i} and has latency constraint L_i. We first consider executing the model M_{k_i} solely in one GPU (Line 1-7 in Algorithm 3.1). Suppose the batch size is b, execution latency of model M_{k_i} is $\ell_{k_i}(b)$, and the worst case latency for any given request is $2\ell_{k_i}(b)$, as we explained in Section 3.2.1. Denote batch size B_i as the maximum value for all b that holds the constraint $2\ell_{k_i}(b) \leq L_i$. Therefore, the maximal throughput, denoted by T_i, of model M_{k_i} with

Algorithm 3.1 Global Scheduling Algorithm

GlobalSchedule(Sessions)

residue_loads ← {}
for $S_i = \langle M_{k_i}, L_i, R_i \rangle$ in Sessions **do**
    $B_i \leftarrow \text{argmax}_b(2\ell_{k_i}(b) \leq L_i)$
    $T_i \leftarrow B_i / \ell_{k_i}(B_i)$
    let $R_i = n \cdot T_i + r_i$
    assign n GPU nodes to execute M_{k_i} with batch B_i
    residue_loads ← residue_load $\oplus \langle M_{k_i}, L_i, r_i \rangle$
for $\langle M_{k_i}, L_i, r_i \rangle$ in residue_loads **do**
    $b_i \leftarrow \text{argmax}_b(\ell_{k_i}(b) + b/r_i \leq L_i)$
    $d_i \leftarrow b_i / r_i$
    $occ_i \leftarrow \ell_{k_i}(b_i)/d_i$
sort residue_loads by occ_i in descending order
nodes ← {}
for $\langle M_{k_i}, L_i, r_i, b_i, d_i, occ_i \rangle$ in residue_loads **do**
    max_occ ← 0
    max_node ← *NULL*
    for $n = \langle b, d, occ \rangle$ in nodes **do**
        $n' \leftarrow$ MergeNodes(n, $\langle b_i, d_i, occ_i \rangle$)
        if $n' \neq$ *NULL* **and** n'.occ > max_occ **then**
            max_occ ← n'.occ
            max_node ← n'
    if max_node ≠ *NULL* **then**
        replace max_node for its original node in nodes
    else
        nodes ← nodes $\oplus \langle b_i, d_i, occ_i \rangle$

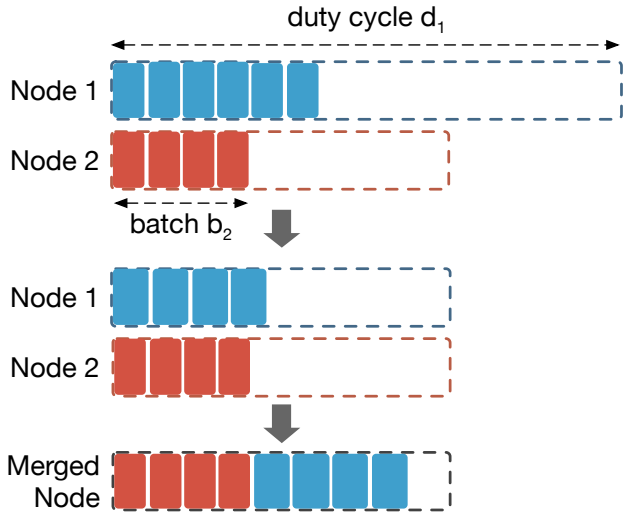


Figure 3.10: Process of merging two nodes. Use the smaller duty cycle as new duty cycle for both nodes. Update the batch size accordingly and re-estimates the batch execution latency. If sum of exec latency doesn't exceed new duty cycle, two nodes can be merged into one.

latency constraint L_i on a single GPU is $B_i/\ell_{k_i}(B_i)$. The number of GPU nodes we allocate to execute just S_i requests is $n = \lfloor R_i/T_i \rfloor$. Note that $n = 0$ for sessions that don't have sufficient requests to utilize an entire GPU.

As a consequence, there will be a residual unallocated load for session S_i after taking into account this allocated load. The next phase assigns this residual load to one other node in the system (Line 8-11 in Algorithm 3.1). Denote $r_i = R_i - n \cdot T_i$ as the request rate of residue load. Suppose we execute the residual load with batch size b, the duty cycle for gathering b inputs d is b/r_i. Then, the worst case latency is $d + \ell_{k_i}(b)$. Therefore, we have the constraint:

$$d + \ell_{k_i}(b) = b/r_i + \ell_{k_i}(b) \leq L_i \tag{3.1}$$

We begin residual load scheduling by choosing for session S_i with residual load r_i the maximum batch size b_i that satisfies the above constraint. Correspondingly duty cycle d is also at its maximal value. Note that this batch size maximizes GPU efficiency for the given latency constraint due to the following argument. Denote *occupancy* (*occ*) as the fraction of the duty cycle d occupied by S_i's residual load invocations: $occ_i(b) = \ell_{k_i}(b)/d$.

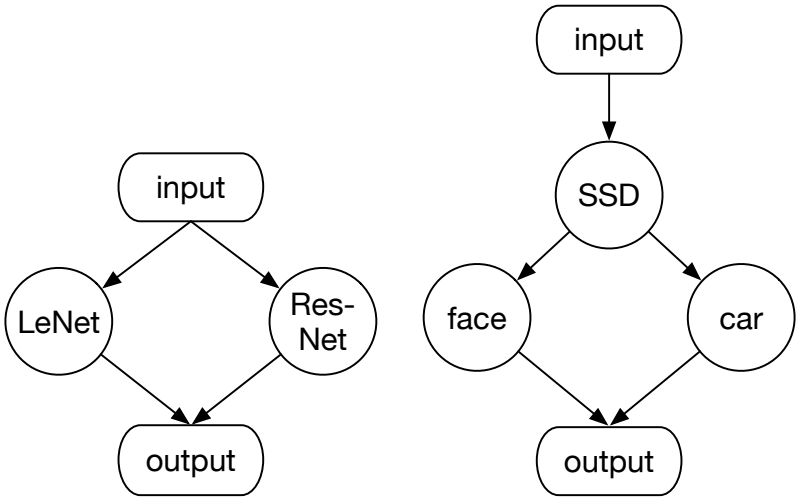
Next, we start to merge these fractional nodes into fewer nodes (Lines 12-25 in Algorithm 3.1). This part resembles the classic bin packing algorithm that first sorts sessions by decreasing occupancy and merges two nodes on to a single node by best fit. The primary difference is how to determine whether two nodes can be merged such that all sessions won't violate their latency SLOs. Figure 3.10 depicts the process to merge two nodes. Suppose we have two sessions S_1 and S_2, with request rates r_1 and r_2, assigned batch sizes b_1 and b_2, and duty cycles d_1 and d_2. We use $d = \min(d_1, d_2)$ as the new duty cycle since d_i is the maximum duty cycle allowed for each session. Without loss of generality, we assume $d = d_2$. We then use $b_1' = d \cdot r_1 \leq b_1$ as the new batch size for executing session S_1. Note that the worst-case latency of requests in session S_1 now becomes $d + \ell_{k_1}(b_1') \leq d_1 + \ell_{k_1}(b_1) \leq L_i$, and we won't violate the latency constraint for S_1 by this adjustment. If $\ell_{k_1}(b_1') + \ell_{k_2}(b_2) \leq d$ and memory capacity permits, a single node can handle the computation of both S_1 and S_2, and we allocate these two sessions to the same node. While the above discussion considers merging two sessions, the underlying principle generalizes to the situation where a session is merged with a set of sessions executing on a given node.

Finally, we extend the algorithm to be incremental across epochs, thus minimizing the movement of models across nodes. If the overall workload decreases, the scheduler attempts to move sessions from the least utilized backends to other backends. If a backend no longer executes any session, the scheduler reclaims this backend and relinquishes it to the cluster manager. If workload increases such that a backend becomes overloaded, we evict the cheapest sessions on this backend until it is no longer overloaded. We then perform bin packing again to place these evicted sessions to other backends.

3.4.2 Scheduling Complex Queries

We now present the query analysis algorithm that operates on dataflow representations of application queries in order to determine the latency SLO splits for the constituent models. The output of this analysis is given as input to the scheduling algorithm of Section 3.4.1 that works with individual models.

The query analysis algorithm extracts the dataflow representations from the application code



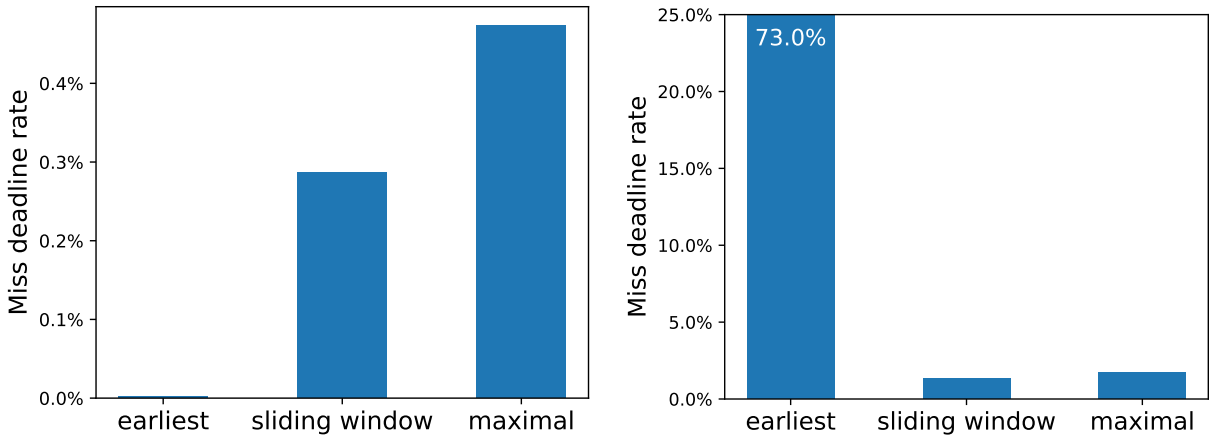
(a) Game (b) Traffic

Figure 3.11: Dataflow representations of two example applications.

(e.g., the left one depicted in Figure 3.11 corresponding to Figure 3.9) and then compute the stage number of each constituent model as the maximum depth of the model invocation with respect to the input. For example, in Figure 3.11a, both the LeNet and ResNet models have a stage number of 1, and in Figure 3.11b, the SSD model is at stage 1 while the face and car recognition models are in stage 2. Models at the same stage will share the same latency SLO.

In addition, we also collect profiling information from the runtime execution of the query. For each edge in the dependency graph $M_i \rightarrow M_j$, the frontends collect information regarding how many instances of M_j is invoked by a single M_i instance. This information is reported to the global scheduler, which aggregates the information.

We now define the optimization objective for the analysis. Suppose there are k models in the query, $M_i (1 \leq i \leq k)$ and that M_i is associated with a latency SLO of L_i, with models at the same stage having the same SLO. Further, for a dependency edge $M_i \rightarrow M_j$, let α_{ij} indicate the number of times M_j is invoked by an instance of M_i. Also, let N_i be the number of GPUs allocated for model M_i. Then, we have the constraint that $\alpha_{ij} N_i T_i(L_i) = N_j T_j(L_j)$, where T_i is the max throughput of model M_i given the latency SLO L_i. Given these constraints and the target throughput associated with the top-level model, we can compute the GPU allocations for a latency split plan. We then



(a) Uniform workload

(b) Poisson workload

Figure 3.12: Compare the percent of requests that miss their deadlines under three batching policies with uniform and poisson workload. The model used in two experiments is Inception [121] with latency SLO 100ms, and the mean request rates for both workload are 1300 requests/sec.

choose the latency split plan that minimizes $\sum N_i$. For queries with only a few stages, the analysis tool just uses brute force to scan all possible splits and return the best; otherwise, it uses simulated annealing [42] to search for the best latency SLO split plan within time limit.

3.5 Runtime

In this section, we briefly describe some of the runtime mechanisms that control the execution of DNN tasks on Nexus frontend and backend nodes. Adaptive batching and GPU multiplexing describe how backend servers handle fluctuant incoming workload and support for multiple models on single GPU. We then compare a few load balancing policies used in frontend servers.

3.5.1 Adaptive Batching

During the serving, it's unlikely that a backend receives the same number of requests as expected batch size at each round of execution, due to either the variation of workload or imperfect load balancing. It's necessary for backends to adjust batch sizes when it receives fewer or more requests than expected batch size. To systematically study this problem, we explore three batching policies

from more conservative policy to more aggressive policy:

- Earliest first: ensures the earliest request in the queue to be processed within its latency SLO. Therefore, the max allowable batch size should be the batch size of which execution time is not longer than the left time budget of the earliest request. This is also the policy used in Clipper [31].

- Sliding window: uses a sliding window of the batch size specified by the global scheduler to scan through the queue. It stops at the first request that won't exceed its deadline under such batch size and drops earlier requests.

- Maximal window: similar to sliding window, but selects maximal possible window size, regardless of the suggested batch size by the global scheduler, such that the batch execution time won't exceed any deadline of requests in the batch, and drops the others.

To compare these three policies, we use simulation and evaluate the percent of requests that miss its deadline under the same workload. We use Inception [121] model with latency SLO 100ms, and send the requests at the mean rate of 1300 reqs/sec. We generate workload using uniform and Poisson distribution. Figure 3.12 depicts the miss deadline rate of three policies mentioned above. We can see that earliest first policy performs best under uniform workload because it is the most conservative policy and prioritizes to not drop requests while sliding window and maximal window policies aim for higher efficiency and throughput and therefore could drop requests prematurely. However, under Poisson distribution, earliest first policy misses the deadline for 73% of requests, significantly higher than sliding window policy (1.3%) and maximal window policy (1.7%).

To understand why earliest first policy performs so bad, we take a closer look by plotting the batch sizes used at each round by these three policies, shown in Figure 3.13. It reveals that earliest first policy uses very small batch sizes after first a few rounds since it needs to guarantee the earliest request not to exceed its deadline. This leads to low efficiency on GPU and therefore low throughput and higher miss rate. On the other hand, maximal window policy constantly uses batch sizes larger than the suggested batch size and is too aggressive to drop requests that could have been served

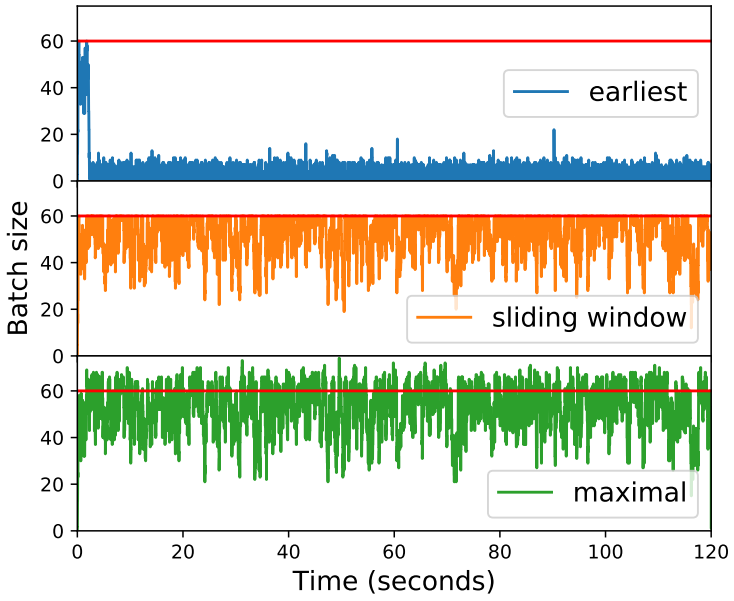


Figure 3.13: Batch sizes used at each round by three batching policies. Red line indicates the batch size set by the global scheduler.

within its deadline. Thus we are seeing higher miss rate of maximal window policy than that of sliding window policy on both workloads. So in Nexus, we use sliding window policy as our adaptive batching policy.

3.5.2 GPU Multiplexing

DNN frameworks provide no specific support for the concurrent execution of multiple models. For example, if two models that share a GPU execute in two processes or containers, they will independently issue requests to execute layers to the underlying GPU (DNN libraries typically require models to be presented as individual layers as *execution kernels*). The GPU runtime will typically serve these requests in first-come-first-served fashion, usually resulting in an arbitrary interleaving of the layers from the two models. The interleaving increases the execution latency of both models and makes it hard to predict the latency. Instead, Nexus manages the execution of all models, so it is able to pick batch sizes and execution schedules for all models in a round-robin fashion to make sure each model abide by their latency SLOs.

Another important observation is that transfer learning [41, 137] adapts a model from one dataset to another or from one task to another by only re-training last one or a few layers. DNN frameworks assume that if models differ in any layer, they cannot be executed in a batched fashion at all. However, in the common setting of model specialization, several models may differ only by their output layer. Batching the execution of all but the output layer can yield substantial batching gains. Nexus automatically recognizes models that have common prefixes, and splits the models into "common prefix" and "different suffix" parts. Backend executes the common parts in a batched manner and different suffix parts sequentially to complete execution.

3.5.3 Load balancing

Load balancing is important to achieve high efficiency in Nexus because of batch execution at the backends. If frontends send more requests than expected batch size to a backend, the exceeded requests need to wait for one round of batch execution before they get executed, and therefore are likely to violate their latency SLOs. On the other hand, if frontends send fewer requests than desired batch size, it will result in lower efficiency since batch size becomes smaller. In essence, frontends need to forward requests to backends according to their desired batch sizes within a very short time of period, usually within tens or hundreds ms. This problem becomes more challenging when the desired batch size is small, due to large models or tight latency SLO, as it leaves less room for imperfect load balancing.

We consider three load balancing mechanism: (a) centralized frontend that sends a batch to backend each time such that batch size suits the best; (b) similar to Sparrow [99], samples two random choices of backend and forwards the request to the backend with shorter queue; (c) randomly selects a backend based on its throughput. In the implementation of two random choices mechanism, we cache the queue sizes at frontend for 10ms to avoid too much overhead by querying backends. We compare these three mechanisms with one frontend server and four backend servers serving SSD model [89]. Table 3.6 lists the desired batch sizes under each latency SLOs and maximum throughput achieved using different load balancing mechanism. We can see that when latency SLO and batch size is small, weighted random algorithms perform significantly worse than centralized frontend due to

Latency SLO (ms)	Batch size	Mechanism	Throughput (req/s)	Relative Throughput (%)
200	3	Centralized	142	100%
		Two choices	123	86.6%
		Weighted random	109	76.8%
400	7	Centralized	191	100%
		Two choices	190	99.5%
		Weighted random	171	89.5%
600	11	Centralized	204	100%
		Two choices	204	100%
		Weighted random	194	95.1%

Table 3.6: Compare the throughput among three load balancing policies for SSD model with different latency SLOs.

unbalanced load within a short time. Two choices mechanism also falls short of centralized frontend under batch size 3. This is mostly caused by using stale backend queue sizes from cache, but this overhead becomes negligible when latency SLO becomes larger. Centralized frontend consistently achieves the highest throughput among the three. However, it is nontrivial to extend such design to a distributed version. Therefore, frontend may become the bottleneck when throughput is very high. We leave the extension to the future work. Overall, we implement all three load balancing mechanisms in Nexus, and use two choices by default.

3.6 Evaluation

We implemented Nexus in C++ with 10k lines of code. Nexus supports the execution of models trained by various frameworks including Caffe [70], Caffe2 [43], Tensorflow [9], and Darknet [109]. Nexus can be deployed in a cluster using Docker Swarm [39] or Kubernetes [49].

To evaluate Nexus, we first use two case studies of real-world applications to compare the end-to-end performance against Clipper and Tensorflow serving (denoted as TF serving below). We then perform a set of experiments to quantify the performance gains of each optimization feature in Nexus. The experiments are performed on a cluster of 16 Nvidia GTX 1080Ti GPUs.

3.6.1 Methodology

For any system and any given workload, we measure the maximum request processing rate such that 99% of the requests are served within their latency SLOs and refer to this measurement as the system's maximum throughput. This metric reflects the GPU efficiency that a system can achieve on a given workload.

Note that neither Clipper nor TF serving provides a cluster-wide solution. We provide a *batch-oblivious scheduler* as a baseline for both Clipper and Tensorflow serving. It works as follows. We first profile the maximum throughput achieved by Clipper or TF serving on a single node for a given model and latency SLO. GPU shares for each model session is calculated to be proportional to its request rate and inversely proportional to the maximum model throughput measured in the previous step. We then use a greedy algorithm to allocate models to GPUs according to their GPU shares. For complex queries, since Clipper and TF serving does not provide functionality for analyzing query performance, we evenly split the latency SLO across the different stages and use this latency split plan in our profiling step. We also use the following node-level configurations to instantiate Clipper and TF serving.

- **Clipper:** Clipper encapsulates each model in a docker container. If the batch-oblivious resource allocator assigns multiple models to a single GPU, we just launch multiple docker containers on this GPU. The Clipper frontend provides a load balancing mechanism. Therefore, we rely on Clipper to load balance requests to multiple replicas of a model.
- **TF serving:** We cannot specify a latency SLO in TF serving. Instead, we pick a maximum batch size for each model based on its profile, so that TF serving doesn't violate its latency SLO. TF serving doesn't provide a frontend to load balance the requests to the replicas either. We therefore built a frontend for TF serving and dispatched the requests to backends according to the mapping generated by the resource allocator.

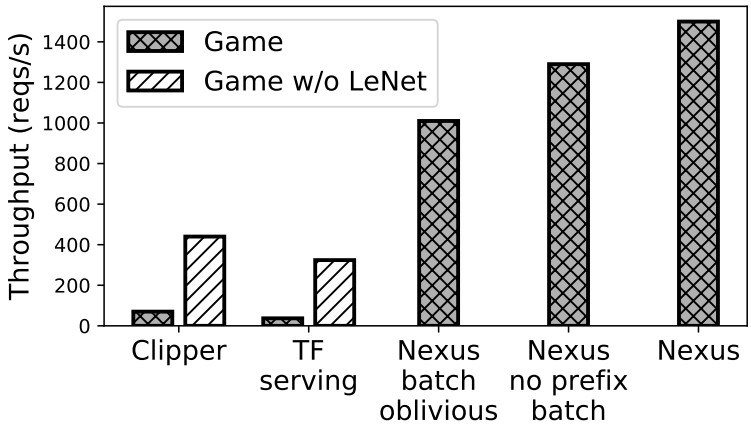


Figure 3.14: Compare the throughput of Nexus, Clipper, and TF serving for 20 live game analysis applications on 16 GPUs. "Nexus batch oblivious" measures throughput of Nexus which uses batch-oblivious scheduler. "Nexus no prefix batch" basically disables prefix batching in Nexus.

3.6.2 Case Study

3.6.2.1 Live Game Analysis

In this case study, we evaluate the application that analyzes frames from live game streams. On each frame, we invoke LeNet [80] 6 times to recognize digits and ResNet-50 [59] once to recognize the icon. ResNet is specialized to each game by re-training the last layer of the model, while all games share the same LeNet. We include 20 games in the case study, and consider a latency SLO of 50ms. Each game receives a different number of concurrent streams. The popularity of 20 games follow the Zipf-0.9 distribution. We perform this experiment on 16 GPUs. During the experiment, we noticed that both Clipper and TF serving performs poorly when the query invokes the tiny LeNet model, as the batch-oblivious scheduler allocates too many GPUs for LeNet. Therefore, we also evaluated Clipper and TF serving on a modified game application that does not invoke LeNet but rather uses all 16 GPUs for ResNet.

Figure 3.14 depicts the total throughput of 20 games using Clipper, TF serving, and Nexus. We include two variants of Nexus to tease out the benefits obtained from some of our techniques: (a) a variant of Nexus that disables prefix batching and uses the batch-oblivious scheduler to perform resource allocation, and (b) a variant of Nexus that disables prefix batching. Overall, Nexus achieves

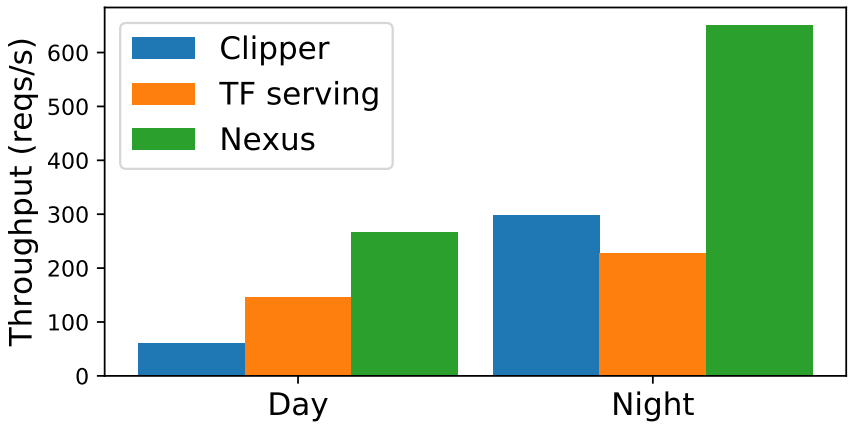


Figure 3.15: Throughput for traffic analysis applications of Clipper, TF serving, and Nexus on 16 GPUs for workload of daytime and nighttime.

21× and 41× the throughput compared to TF serving and Clipper respectively on the original game application. Moreover, TF serving and Clipper can process only 3-4 times fewer requests even after we remove LeNet from the application. Comparing across different Nexus variants, our global scheduler achieves 27% higher throughput compared to the batch-oblivious scheduler, and prefix batching bumps up throughput by an extra 16%.

3.6.2.2 Traffic Monitoring

The traffic monitoring application analyzes live video streams from many traffic cameras. It first detects objects using SSD [89] and recognizes the make and model of cars and faces using GoogleNet-car [135] and VGG-Face [101], respectively. Figure 3.9 shows the query code for the traffic app. The latency SLO of this query is 400ms. We measure the throughput for this application using traffic feeds obtained during both daytime and nighttime. Figure 3.15 shows that Nexus achieves 1.8× (day) and 2.9× (night) throughput compared to TF serving, and 4.4× (day) and 2.2× (night) throughput compared to Clipper. Nexus is able to adapt the latency split for the SSD model from 302ms during daytime to 345ms at night because fewer cars and pedestrians appear in the frames. Therefore, we see a greater increase in throughput of Nexus at night than that of TF serving.

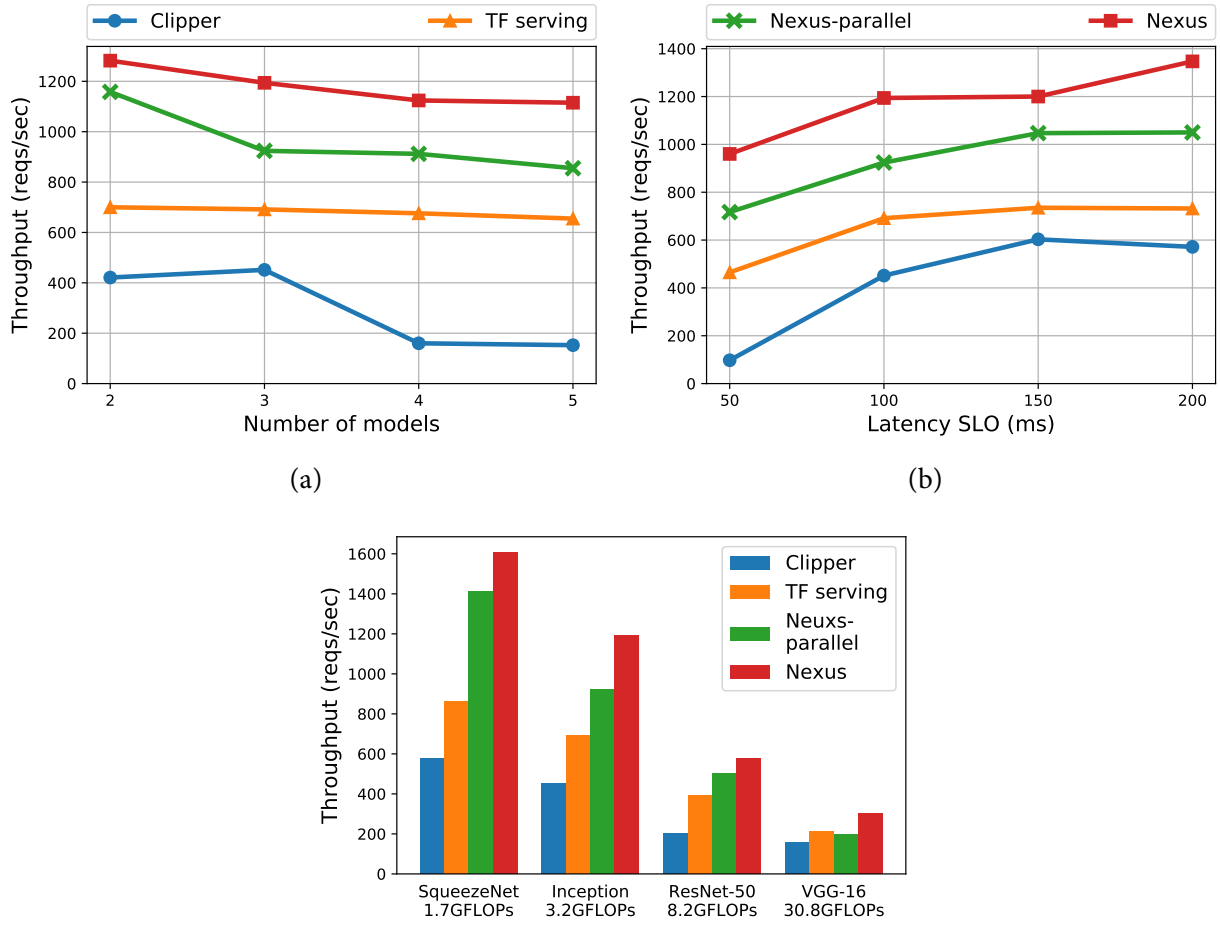


Figure 3.16: Compare the throughput when having multiple models on a single GPU among Clipper, Tensorflow Serving, Nexus variant which executes models in parallel (Nexus-parallel), and standard Nexus, which executes models in round-robin manner. (a) compares the throughput of Inception models under different number of models on one GPU where latency SLO is 100ms. (b) compares the throughput of Inception models under different latency SLO when there are 3 models on one GPU. (c) compares the throughput of different model architecture where latency SLO is 100ms and 3 models on one GPU.

3.6.3 Microbenchmark

We perform several microbenchmarks to evaluate the throughput improvement for each feature used in Nexus.

3.6.3.1 Single GPU performance

On a single GPU, we compare the performance of Nexus, Clipper, and Tensorflow Serving for supporting multiple models on a single GPU. In addition, Nexus performs prefix batching when it detects different DNN models share common prefix to further improve the throughput.

GPU Multiplexing. Figure 3.16 compares the throughput when executing multiple models on a single GPU. Three factors affect the performance of GPU multiplexing: number of models that share the GPU, latency SLO of models, and the model architecture. By default, we use the Inception [121] model, latency SLO of 100ms, and deploy 3 models on the same GPU. We include one variant of Nexus in this experiment, denoted as Nexus-parallel, which executes models in parallel to quantify how much GPU interference affects the throughput. Figure 3.16a compares the throughput ranging from 2 models to 5 models on one GPU. The results show that Nexus achieves 10%–30% higher throughput than Nexus-parallel. Interference becomes more severe when more models contend on the same GPU. Figure 3.16b compares the throughput while varying the latency SLO from 50ms to 200ms. When latency SLO becomes higher, Nexus-parallel tends to achieve higher throughput relatively because there is more slack to tolerate interference. We also repeat the experiment for other model architectures including SqueezeNet [66], ResNet-50 [59], and VGG-16 [117] (shown in Figure 3.16c). We observe that larger models tend to suffer more from interference.

Nexus achieves 1.4–2.1× throughput compared to TF serving, and 1.9–9.8× throughput compared to Clipper on a single GPU. Because Clipper executes a model in a docker container and there is no coordination mechanism among containers, Clipper suffers from interference between models. We can see that Clipper's throughput reduces significantly when more models are allocated on the same GPU or the latency SLO becomes more restricted. TF serving executes multiple models in a round robin way, same as Nexus. But TF serving doesn't overlap the pre- and post-processing in CPU with model execution in GPU. TF serving also waits for a target batch size to be filled up until a timeout. Both cause the GPU to be idle and correspondingly harms the throughput.

Prefix Batching. Figure 3.17 evaluates prefix batching against executing the models separately on one GPU. Figure 3.17a depicts the throughput improvement by varying the number of Inception

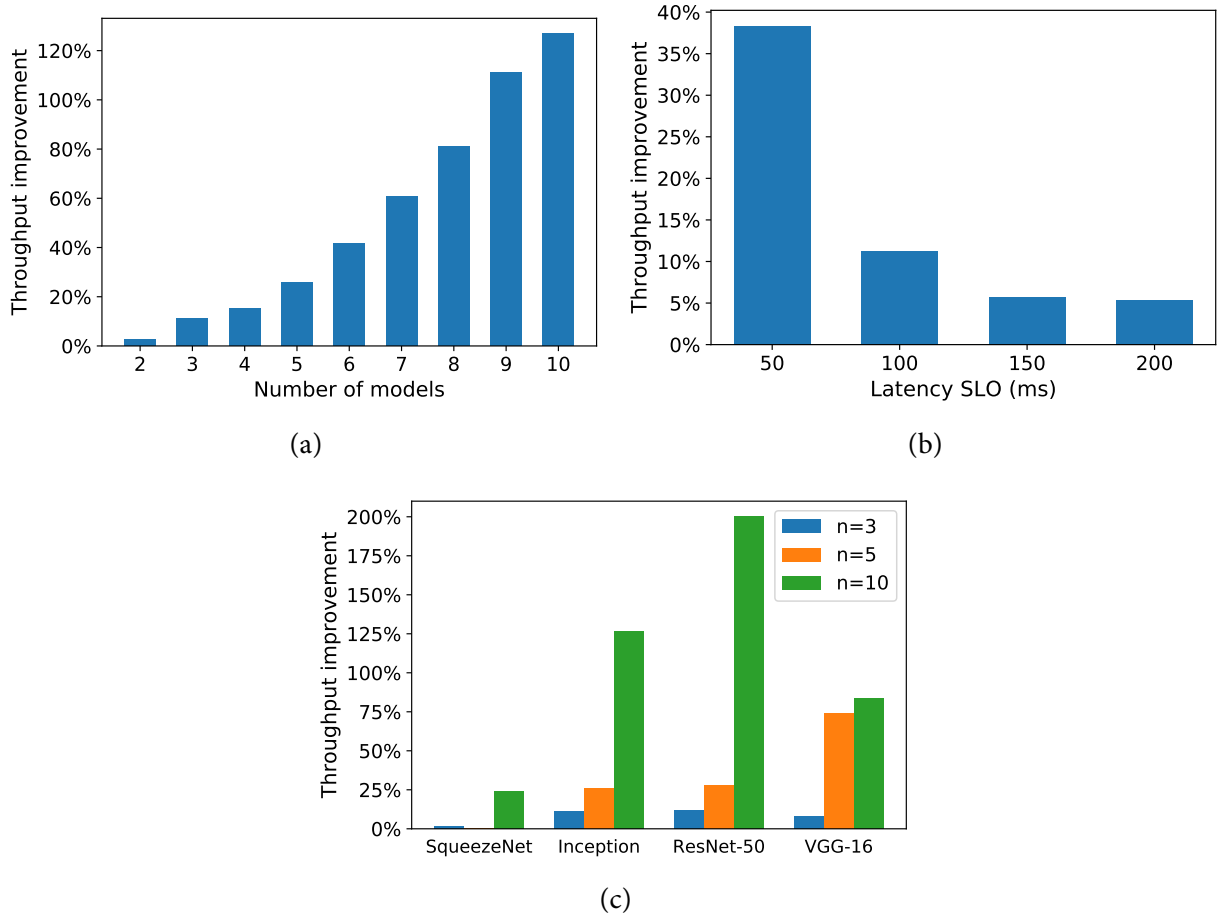


Figure 3.17: Compare the throughput improvement of batching the common prefix of model variants against running these models separately on one GPU. (a) and (b) both use Inception model. (a) compares throughput improvement for different number of models that share common prefix with latency SLO 100ms, whereas (b) compares throughput improvement under different latency SLO when 3 models share prefix. (c) shows the throughput improvement for different model architecture under 100ms latency SLO for 3, 5, and 10 models that share prefix.

model variants that differ only in the last layer and with a latency SLO 100ms. It shows that prefix batching improves the throughput by up to 125%. Prefix batching primarily benefits from using a larger batch size for the common prefix; otherwise, models are executed at much smaller batch sizes when more models are multiplexed on to a GPU. Similarly, when latency SLO becomes smaller, prefix batching provides more throughput gains, as shown in Figure 3.17b. Figure 3.17c applies prefix batching on different model architectures when there are 3, 5, and 10 models. Prefix batching doesn't

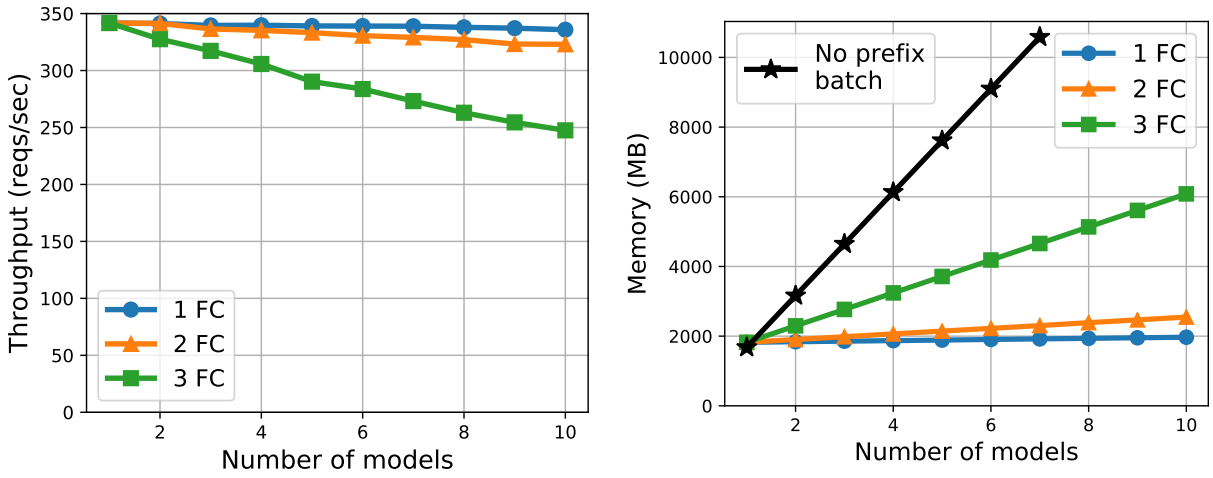


Figure 3.18: Evaluate the overhead and memory use of prefix batching for VGG-16 varying suffix lengths. k-FC means that suffix contains last k fully-connected layers. (a) Throughput of prefix batching for different number of models and suffix length. (b) Memory footage of prefix batching with regard to number of models and suffix length. Black line shows the memory use without prefix batching.

improve the throughput for SqueezeNet when there are 3 or 5 model variants, because it is a much smaller model, and they are able to saturate GPU efficiency even without prefix batching. For other settings, prefix batching improves throughput by up to 3x.

Note that as prefix batching executes different suffixes of model variants sequentially, it introduces certain overheads if the suffix includes heavy-weight computations. Figure 3.18a shows the throughput of prefix-batched models varying the length of the suffix from the last layer to three fully-connected layers and varying the number of models. When the suffix contains no more than two fully-connected layers, execution of suffixes imposes less than 5% overhead even with 10 models. But, because the third from last layer is quite heavy-weight, we can see that prefix batching only achieves 72% of throughput for 10 models compared to no prefix batching. Another benefit brought by prefix batching is memory savings since we only need to allocate one copy for the common prefix. Figure 3.18b reveals that the memory use of prefix batching grows sub-linearly with the number of models, whereas the GPU runs out of memory at 8 models without prefix batching.

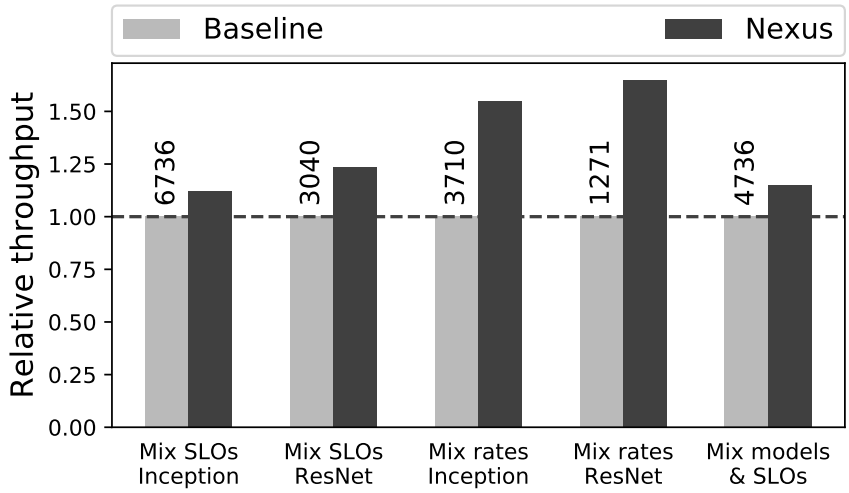


Figure 3.19: Compare the throughput between Nexus and Nexus using batch-oblivious resource allocator.

3.6.3.2 Resource Allocation on Multiple GPUs

We now compare the global scheduler in Nexus against batch-oblivious scheduler. We measure the throughput of standard Nexus and Nexus using batch-oblivious scheduler as baseline. Both need to allocate 16 sessions on 8 GPUs under 5 scenarios: (a) 16 Inception or ResNet models with mixed SLOs ranging from 50ms to 200ms, (b) 16 Inception or ResNet models with mixed request rates following Zipf-0.9 distribution, (c) 8 different model architectures, each associated with two SLOs, 50ms and 100ms. Figure 3.19 depicts the relative throughput of standard Nexus with regard to baseline. Nexus outperforms baseline by 11–64% because our global scheduler is aware of "squishy" performance of batch execution while the batch-oblivious scheduler is prone to overloading a backend by placing excessive workload.

3.6.3.3 Complex Query

To evaluate the performance gain of the query analyzer, we compare the throughput of Nexus with and without the query analyzer. The baseline simply splits the latency SLO evenly across the various stages in the query. The query includes two stages: (a) first stage executes SSD, and then (b) invokes Inception model for α times. The experiment is performed on 8 GPUs. We vary the latency SLO

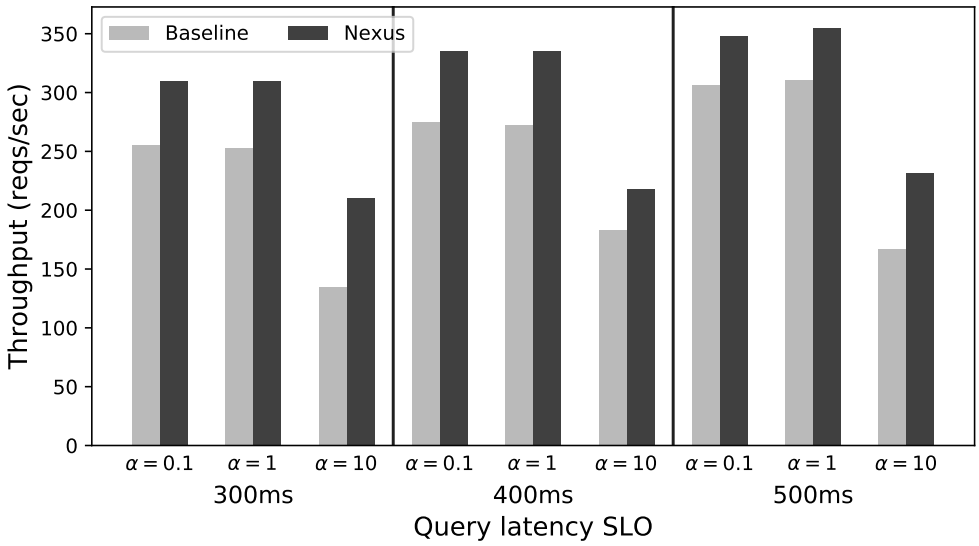


Figure 3.20: Compare the throughput between Nexus with and without query analyzer. The query in this experiment includes two models, SSD and Inception. We vary the latency SLO and α in the query, where α indicates the number of times Inception is invoked by SSD.

from 300ms to 500ms and choose α to be 0.1, 1, and 10. Figure 3.20 shows that Nexus with the query analyzer achieves 13–55% higher throughput than the baseline.

To further understand the benefit of query analyzer, we analyze another complex query that uses SSD in the first stage and VGG-Face in the second stage. We fix the latency SLO to 400ms and α = 1, and explore a wide range of latency split plans, while query analyzer outputs the allocation plan of 345ms for SSD and 55ms for VGG-Face. The experiments are measured on 8 GPUs. Figure 3.21 demonstrates that query analyzer is able to find the best latency allocation strategy as it achieves the highest throughput.

3.6.4 Large-scale deployment with changing workload

We deploy Nexus on a cluster of 64 GTX 1080Ti GPUs and run 10 applications with mixed models and latency SLOs for 15 minutes with changing workload. Figure 3.22 demonstrates that Nexus is able to scale up when workload increases, and consolidate workload and recycle GPUs when workload decreases while maintaining high good rate (percent of requests that are served within latency SLO) for all applications.

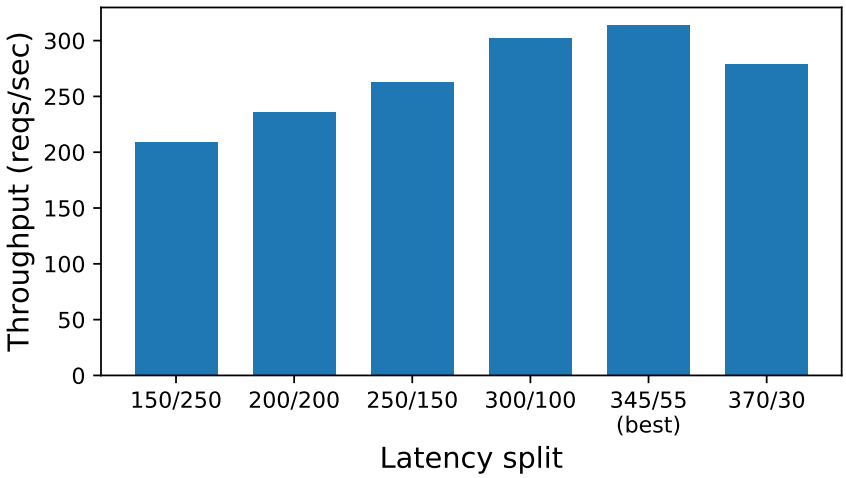


Figure 3.21: Explore different latency split plans for SSD and VGG-Face query where query latency SLO is 400ms and $\alpha = 1$. A/B indicates that latency SLO for SSD and VGG-Face are Ams and Bms respectively. Best split plan 345/55 is generated by query analyzer.

3.7 Conclusion

We proposed a scalable and efficient system design for serving Deep Neural Network (DNN) applications. Instead of serving the entire application in an opaque CPU-based container with models embedded in it, which leads to sub-optimal GPU utilization, our system operates directly on models and GPUs. This design enables several optimizations in batching and allows more efficient resource allocation. Our system is fully implemented, in C++ and evaluation shows that Nexus can achieve 1.4-41× more throughput relative to state-of-the-art baselines while staying within latency constraints (achieving a "good rate") > 99% of the time.

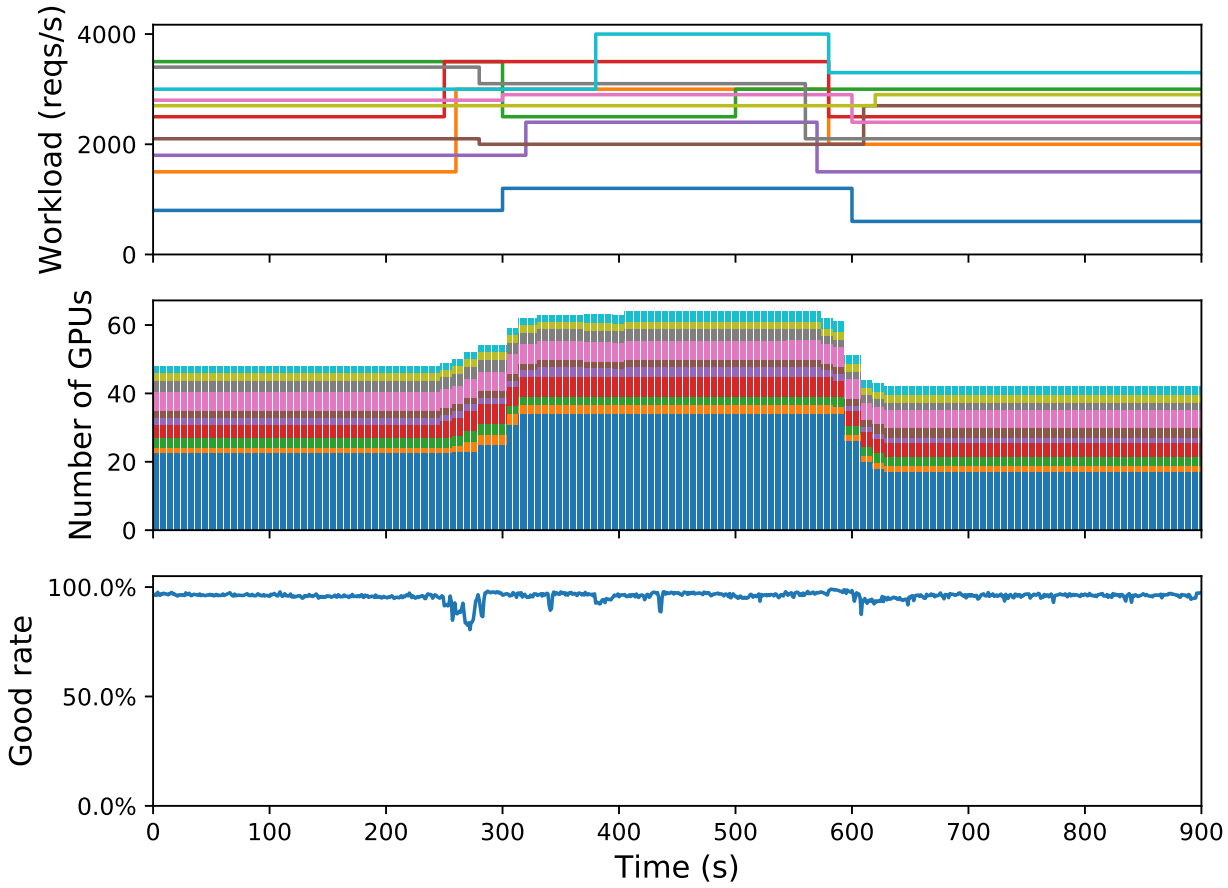


Figure 3.22: Changing workload of 10 applications over time on 64 GPUs. Top row shows the request rates over the time. Middle row shows the number of GPUs allocated to application. Bottom row shows the good rate, percent of requests that are served within latency SLO.

Chapter 4

MCDNN: Approximation-Based Execution Framework for Mobile-Cloud Environment

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

4.1 Background

Continuous mobile vision (CMV) refers to the setting in which a user wears a device that includes a continuously-on (we target ten hours of continuous operation) camera that covers their field of view [50, 104, 124, 106, 15]. The device is often a custom wearable such as Google Glass, but possibly

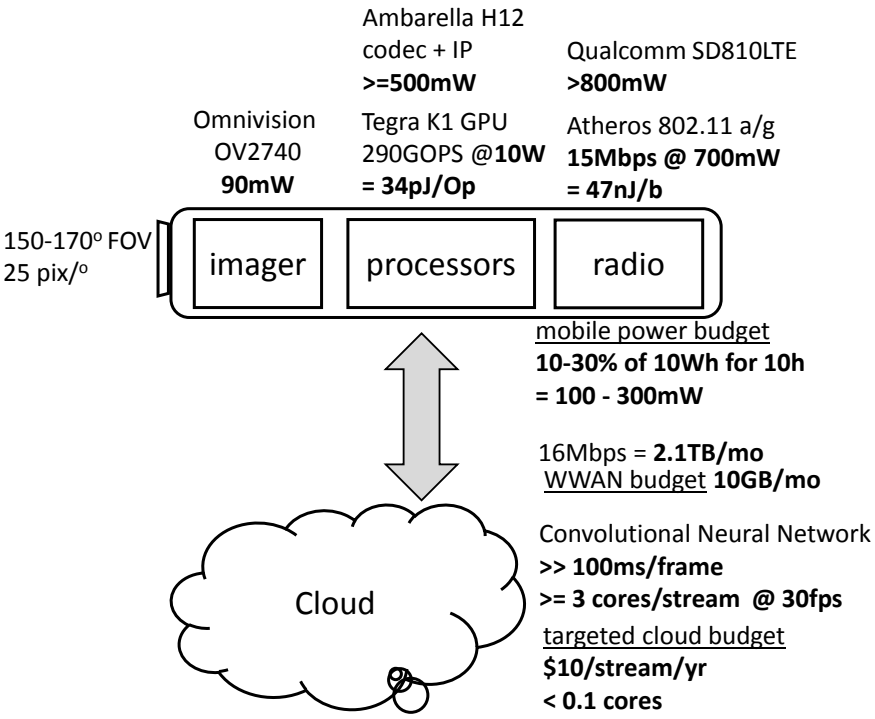


Figure 4.1: Basic components of a continuous mobile vision system.

just a mobile phone in a pocket. Video footage from the camera is analyzed, typically in real time, using computer vision techniques to infer information relevant to the user. While current systems are research efforts aimed at key niche applications such as cognitive assistance for the elderly and navigational assistance for the blind, in the near future we anticipate a *multi-programming* setting aimed at the general consumer, where multiple applications issue distinct queries simultaneously on the incoming stream. For example, various applications may need to recognize what the wearer eats, who they interact with, what objects they are handling as part of a task, the affect of the people they interact with and the attributes of the place they are in. In this section, we introduce the resources involved in such a system and motivate careful resource management via controlling the overhead of Deep Neural Network (DNN) execution.

Video processing itself usually involves some combination of *detecting* regions of interest in each frame (e.g., faces, pedestrians or objects), *tracking* the trajectory of detected regions across time and *recognizing* the detailed identities and attributes of these regions (e.g., recognizing the identity

of people and objects interacted with). Although traditionally detection and tracking have been performed with relatively lightweight algorithms, state-of-the-art variants of these are switching to Deep Neural Networks (DNNs) [85, 76]. More recent work has begun to integrate detection and recognition using DNNs [110]. Since detection and tracking computations are expected to be performed frequently (e.g., a few times a second), we expect DNNs to be applied several times to many of the video frames. We therefore view DNN execution as the bulk of modern vision computation.

Figure 4.1 sketches the architecture of a state-of-the-art mobile/cloud Continuous Mobile Vision (CMV) system. The two main physical components are a battery-powered mobile device (typically some combination of a phone and a wearable) and powered computing infrastructure (some combination of a cloudlet and the deep cloud). The camera on the wearable device must usually capture a large field of view at high resolution and moderate frame rate. A resolution of 4k (4096×2160 pixels) at 15 frames per second is not unreasonable[1], drawing 90mW from a modern imager.

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

In comparison, a large 3Ah mobile phone battery of today yields roughly 1.2W over 10 hours. Further, today's consumer mobile plans cap data use at 10GB per month. Finally, continuous use of the required cores will cost roughly $150-300 per year[2]; more realistic workloads could easily be 10 times as costly. Offloading *all* data for cloud-processing using maximally accurate DNNs would

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

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

thus probably only be practical in usages with a large dedicated wearable battery, lightly subscribed WiFi connection (thus limiting mobility) and substantial cloud budget. One path to reducing these resource demands is to reduce the amount of data transmitted by executing "more lightweight" DNN calculations (e.g. detection and tracking) on the device and only transmitting heavier calculations (e.g., recognition) to the cloud. Once at the cloud, reducing the overhead of (DNN) computations can support more modest cloud budgets.

Now consider performing vision computations on the device. Such local computation may be *necessary* during the inevitable disconnections from the cloud, or just *preferable* in order to reduce data transmission power and compute overhead. For simplicity, let us focus on the former ("purely local") case. Video encoding and communication overhead would now be replaced (at least partially) by computational overhead. Given mobile CPU execution speeds (several seconds of execution latency for standard DNNs), we focus on using mobile GPUs such as the NVIDIA Tegra K1 [6], which supports 300GFLOPS peak at a 10W *whole-system-wide* power draw. We time DNN execution on the K1 to take 100ms (for the "AlexNet" object recognition model) to 900ms (for "VGGNet"). Handling the aforementioned conservative processing rate of 1 DNN computation per frame at 15-30fps would require 1.5-30 mobile GPUs, which amounts to 15-300W of mobile power draw on the Jetson board for the K1 GPU. Even assuming a separate battery, a roughly 1-1.2W continuous power draw is at the high end of what is reasonable. Thus, it is important to substantially reduce the execution overhead of DNNs on mobile GPUs.

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

4.2 Approximate Model Scheduling

As described in Section 2.2, a common theme model optimization is the trading off of resource use for accuracy. When applied to the distributed, streaming, multi-model setting of MCDNN, several related questions present themselves. What is the "best" level of approximation for any model, at any given time, given that many other models must also execute at limited energy and dollar budgets? Where (device or cloud) should this model execute? Do the streaming and multi-programming settings present opportunities for new kinds of optimizations? To clarify the issues involved, we first capture these questions in a formal problem definition below and then describe MCDNN's solution.

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

4.2.1 Fractional packing and paging

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

The standard fractional packing problem [20] seeks to *maximize* a linear sum of variables while allowing upper bounds on other linear sums of the same variables. In AMS, the assumption is that when some model is requested to be executed at a time step, we have the option of choosing to execute an arbitrary "fraction" of that model. In practice, this fraction is a variant of the model that has accuracy and resource use that are fractional with respect to the "best" model. These fractions x_t at each time step t are our variables. Assuming for the moment a linear relation between model fraction and accuracy, we have average accuracy over all time steps proportional to $\sum_t a_t x_t$, where a_t is the accuracy of the best model for request t. Finally, suppose one-time resource-use cost (e.g. energy, cloud cost) is proportional to the fraction of the model used, with e_t the resource use of the best

variant of the model requested at t. A resource budget E over all time steps gives the upper-bound constraint $\sum_t e_t x_t \leq E$. Maximizing average accuracy under budget constraints of this form gives a packing problem.

The weighted paging problem [16] addresses a sequence of t requests, each for one of M pages. Let $x_{mj} \in [0,1]$ indicate what fraction of page m was evicted in between its ith and $i+1$th requests. The goal is to *minimize* over t requests the total number of (weighted) paging events $\sum_{mj} c_t x_{mj}$. At the same time, we must ensure *at every time step* that cache capacity is not exceeded: if k is cache size and R_{mt} is the number of requests for model m up to time t and N_t is the total number of requests in this time, we require $\forall_t N_t - \sum_m x_{mR_{mt}} \leq k$. If x's are once again interpreted as fractional models and c_i represent energy costs, this formulation minimizes day-long energy costs of paging while respecting cache capacity.

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

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

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

Note that we do not model several possibly relevant effects. The "cache size" may vary across timesteps because, e.g., the machine is being shared with other programs. The cost/accuracy tradeoff may change with context, since models are specializable in some contexts. Sharing across models may mean that some groups of models may cost much less together than as separately. Since different

apps may register different models to handle the same input type (e.g., face ID, race and gender models may all be registered to process faces), we have a "multi-paging" scenario where models are loaded in small batches. Although we do not model these effects formally, our heuristic scheduling algorithm is flexible enough to handle them.

4.2.2 AMS problem definition

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

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

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

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

up to (and including) timestep τ. For any timestep τ, let $R(\tau) = \{M_i | n_{i\tau} \geq 1\}$ be the set of models requested up to and including timestep τ.

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

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

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

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

$$\max_x \sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} a_{ix_{ik}} + a_{ix'_{ik}}, \tag{4.1}$$

while keeping total on-device energy use within budget:

$$\sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} e_{ix_{ik}}\delta(|x_{i(k-1)} - x_{ik}| + y_i^{k-1}) + c_{ix_{ik}} \leq E, \tag{4.2}$$

keeping within the cloud cost budget:

$$\sum_{i=1}^{n} \sum_{k=1}^{n_{iT}} d_{ix'_{ik}} \leq D, \tag{4.3}$$

and at each timestep, not exceeding cache capacity:

$$\bigvee_{1\le\tau\le T} \sum_{\substack{M_i\in R(\tau)\\1\le k\le n_{i\tau}}} s_{ix_{ik}} - \sum_{\substack{M_i\in R(\tau)\\1\le\tau'\le\tau}} s_{iy_{i\tau'}} \le S, \tag{4.4}$$

ensuring that the selected model variant executes fast enough:

$$\bigvee_{\substack{1\le i\le n\\1\le k\le n_{iT}}} l_{ix_{ik}} \le t_{\tau_{ik}+1} - t_{\tau_{ik}}, \tag{4.5}$$

and ensuring various consistency conditions mentioned above:

$$\bigvee_{\substack{1\le i\le n\\1\le k\le n_{iT}}} 0 \le x_{ik}, x'_{ik}, y_{ik} \le 1 \text{ and } x_{ik}x'_{ik} = 0 \tag{4.6}$$

4.2.3 Discussion

We seek to solve the above optimization problem online. On the surface, AMS looks similar to fractional packing: we seek to maximize the weighted sum of the variables that are introduced in an online fashion, while upper-bounding various weighted combinations of these variables. If we could reduce AMS to packing, we could use the relevant primal-dual scheme [20] to solve the online variant. However, the AMS problem as stated has several key differences from the standard packing setting:

- Most fundamentally, the paging constraint (Equation 4.4) seeks to impose a cache capacity constraint *for every timestep*. The fractional packing problem strongly requires the number of constraints to be fixed before streaming, and therefore independent of the number of streaming elements.

- The mappings from the fractional size of the model x_{ik} to accuracy ($a_{x_{ik}}$), energy cost ($e_{x_{ik}}$), etc., are left unspecified in AMS. The standard formulation requires these to be linear, e.g, $a_{x_{ik}} = a_i x_{ik}$.

- The standard formulation requires the upper bounds to be independent of timestep. By requiring execution latency at timestep τ_{ik} to be less than the interval $t_{\tau_{ik}+1} - t_{\tau_{ik}}$, AMS introduces a time dependency on the upper bound.

- The mutual exclusion criterion (Equation 4.6) introduces a non-linearity over the variables.

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

4.3 System Design

Solving the AMS problem requires two main components. First, the entire notion of approximate model scheduling is predicated on the claim that the models involved allow a useful tradeoff between resource usage and accuracy. Instead of using a single hand-picked approximate variant of a model, MCDNN dynamically picks the most appropriate variant from its *model catalog*. The model catalog characterizes precisely how model optimization techniques of Section 2.2 affect the tradeoffs between accuracy and model-loading energy on device (e_{ix}; Equation 4.2), model-execution energy on device (c_{ix}; Equation 4.2), model-execution dollar cost on cloud ($d_{ix'_{ik}}$; Equation 4.3), model size ($s_{ix_{ik}}$; Equation 4.4) and model execution latency (l_{ix}; Equation 4.5). Below, we provide a detailed measurement-based characterization of these tradeoffs. Also, we introduce two new model optimization techniques that exploit the streaming and multi-programming setting introduced by MCDNN. The second component for solving AMS is of course an online algorithm for processing model request streams. We present a heuristically motivated scheduling algorithm below. Finally, we describe briefly the end-to-end system architecture and implementation that incorporates these components.

4.3.1 Model catalogs

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

Task	**Description** (# training images, # test images, # class)
V	VGGNet [117] on ImageNet data
A	AlexNet on ImageNet data [37] for object recognition (1.28M, 50K, 1000)
S	AlexNet on MITPlaces205 data [143] for scene recognition (2.45M, 20K, 205)
M	re-labeled S for inferring manmade/natural scenes
L	re-labeled S for inferring natural/aritificially lighting scenes
H	re-labeled S with Sun405 [131] for detecting horizons
D	DeepFaceNet replicating [123] with web-crawled face data (50K, 5K, 200)
Y	re-labeled D for age: 0-30, 30-60, 60+
G	re-labeled D for gender: M, F
R	re-labeled D for race: African American, White, Hispanic, East Asian, South Asian, Other

Table 4.1: Description of classification tasks.

described in Section 2.2. MCDNN applies the following traditional techniques:

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

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

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

The individual impact of these optimizations have been reported elsewhere [75, 55, 117]. We focus here on understanding broad features of the tradeoff. For instance, does the accuracy plunge with lowering of resources or does it ramp down gently? Do various gains and trends persist across a large

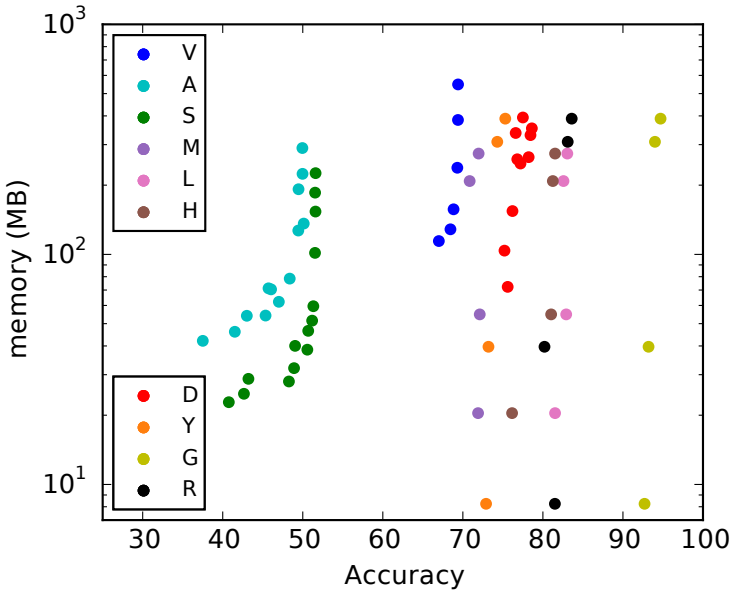


Figure 4.2: Memory/accuracy tradeoffs in MCDNN catalogs.

variety of models? How does resource use compare to key mobile budget parameters such as wireless transmission energy and commercial cloud compute costs? How do various options compare with each other: for instance, how does the energy use of model loading compare with that of execution (thus informing on the importance of caching)? Such system-level, cross-technique questions are not typically answered by the model-optimization literature.

To study these questions, we have generated catalogs for ten distinct classification *tasks* (a combination of model architecture and training data, the information a developer would input to MCDNN). We use state-of-the art architectures and standard large datasets when possible, or use similar variants if necessary. The wide variety of tasks hints at the broad utility of DNNs, and partially motivates systems like MCDNN for managing DNNs. Table 4.1 summarizes the classification tasks we use. For each model in the catalog, we measure average accuracy by executing it on its validation dataset, and resource use via the appropriate tool. We summarize key aspects of these catalogs below.

Figure 4.2 illustrates the memory/accuracy tradeoff in the MCDNN catalogs corresponding to the above classification tasks. For each classification task, we plot the set of variants produced by the model above optimization techniques in a single color. MCDNN generated 10 to 68 variants

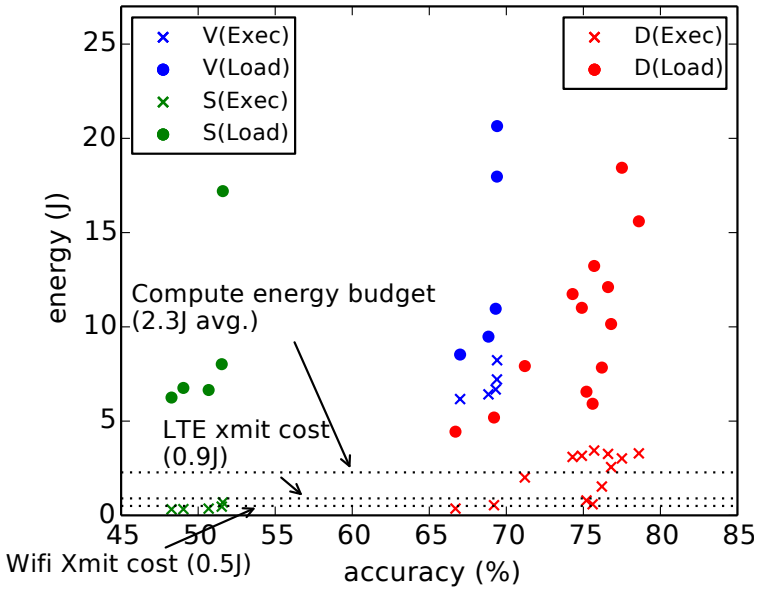


Figure 4.3: Energy/accuracy tradeoffs in MCDNN catalogs.

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

Figure 4.3 illustrates energy/accuracy tradeoffs for the face, object and scene recognition tasks. Energy numbers are measured for an NVIDIA Tegra K1 GPU on a Jetson board using an attached DC power analyzer [5]. The figure shows both execution (crosses) and load (dots) energies for each model variant. Once again, across tasks, model optimizations yield significantly better resource use and modest loss of classification accuracy. Further, the falloff is gentle, but not as favorable as for memory use. In scene recognition, for instance, accuracy falls of relatively rapidly with energy use, albeit from a very low baseline. The horizontal dotted lines indicate the energy budget (2.3J)

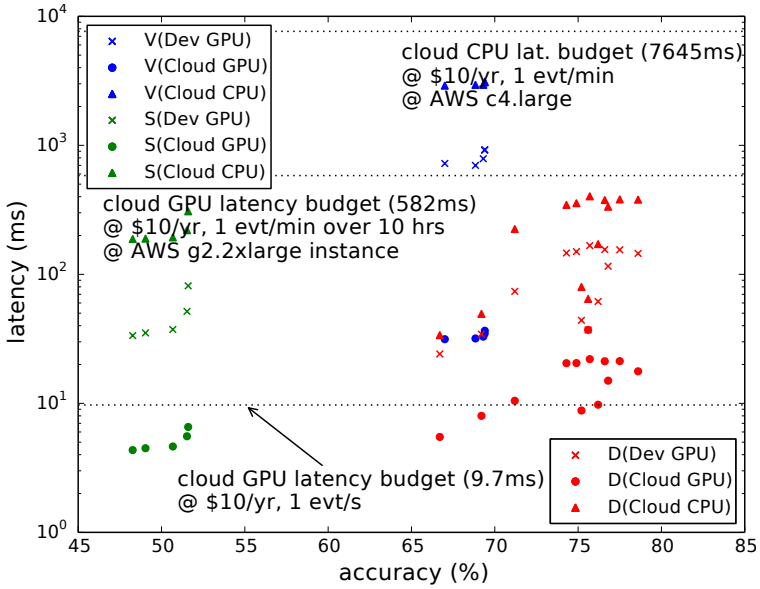


Figure 4.4: Latency/accuracy tradeoffs in MCDNN catalogs.

to support 5 events/minute with 25% of a 2Ah battery, to transmit a 10kB packet over LTE (0.9J), and over WiFi (0.5J) [25]. Note the packet transmit numbers are representative numbers based on measurements by us and others, but for instance LTE packet transmit overhead may be as high as 12J for a single packet [63], and a 100ms round trip over a 700mW WiFi link could take as little as 0.07J. For the most part, standard model optimizations do not tip the balance between local execution and transmission. If execution must happen locally due to disconnection, the mobile device could support at most a few queries per minute. Finally, as the colored dots in the figure show, the energy to load models is higher by 2-5× than to execute them: reducing model loading is thus likely a key to power efficiency.

Figure 4.4 illustrates latency/accuracy tradeoffs for the face, object and scene recognition tasks. The figure shows time taken to execute/load models on devices (crosses) and on the cloud (dots). The measurements shown are for GPUs on-device (a Tegra K1) and on-cloud (an NVIDIA k20), and for cloud-based CPUs. The dotted lines again indicate some illuminating thresholds. For instance, a cloud-based GPU can already process one event per second on a $10/year budget on a cloud GPU (which translates to 9.7ms per event) for scene recognition and some variants of face recognition, but not object recognition. A cloud-based CPU, however, can process one event per minute but not one

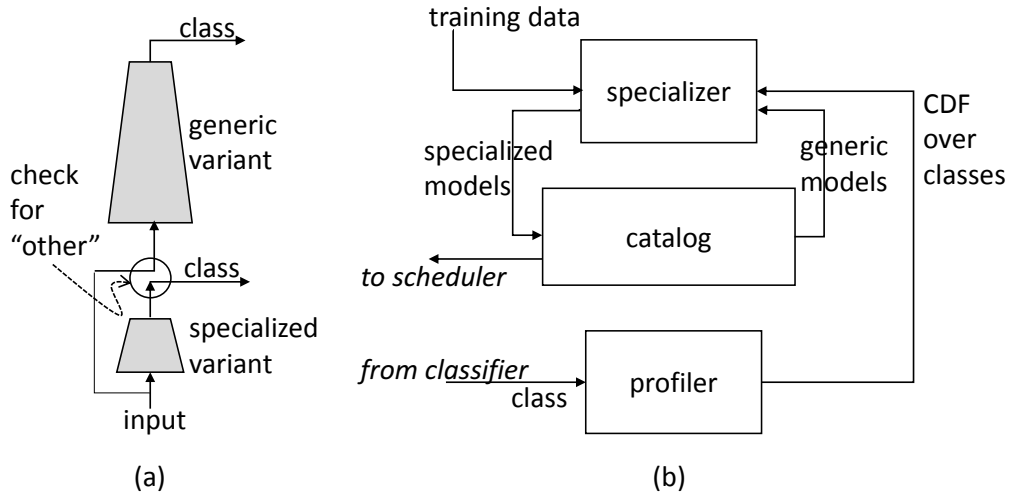


Figure 4.5: Model specialization: (a) Cascading specialized models. (b) MCDNN infrastructure for specialization.

per second (roughly 130ms budget, not shown) at this budget. The efficacy of model optimization carries over to execution and loading speed.

4.3.2 System support for novel model optimizations

The streaming, multi-model setting is a relatively uncommon one for model optimization. We therefore adopt two new optimizations, sequential specialization (described in Chapter 5) and computation sharing to exploit streaming and multi-model respectively. We desribe the system supports to enable these two optimizations as follows.

4.3.2.1 Support for sequential specialization

Remind that sequential specialization adopts a cascaded approach (Figure 4.5(a)) to exploit the temporal locality. 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 4.5(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.

4.3.2.2 Support for model 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 4.6(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

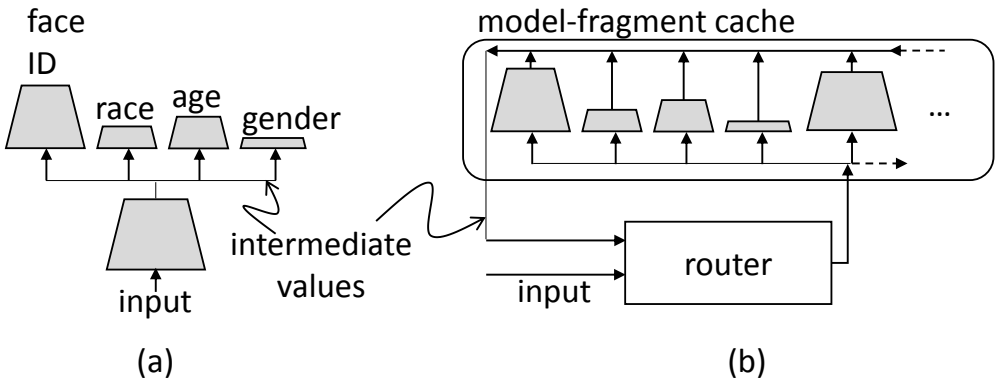


Figure 4.6: Model sharing: (a) Sharing model fragments for facial analysis. (b) MCDNN infrastructure for sharing, replicated in the client and cloud.

s_l are from the library and intended for sharing. Let m_l be the trained version of s_l, also available pre-trained from the library. The compiler assembles a trained model $m_l + m_u$ consisting of two *fragments*. The lower fragment is m_l. The upper fragment, m_u, is a *freshly trained* variant of s_u, trained on $t' = m_l(t), v' = m_l(v)$, i.e., the training "input" to the upper fragment is the result of running the original training input through the lower fragment. The compiler records the fragments and their dependence in the catalog passed to the runtime. **Note that since the upper shared fragment needs to be re-trained sharing is not simply common-subexpression elimination on DNNs**.

Figure 4.6(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.

4.3.3 MCDNN's approximate model scheduler

When applications are installed, they register with the *scheduler* a map from input types to a catalog of model fragments to be used to process those inputs, and handlers to be invoked with the result from each model. The catalog is stored on disk. When a new input appears, the scheduler (with help from

the router and profiler) is responsible for identifying the model variants that need to be executed in response, paging if needed the appropriate model variants in from disk to the in-memory *model-fragment cache* in the appropriate location (i.e., on-device or on-cloud) for execution, executing the models on the input and dispatching the results to the registered handlers.

This online scheduling problem is challenging because it combines several elements considered separately in the scheduling literature. First, it has an "online paging" element [16], 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" [20] 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 [22]. 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 4.1 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 12) and *which* variant to execute (Line 13). The variant and location selected are the ones with the maximum accuracy (Line 13) under estimated future resource (energy on the device, and cash on the cloud) use (Lines 10,11).

To estimate future resource use, for model n (Lines 15-21), 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

Algorithm 4.1 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: **return** r

▷ Update the cache by inserting the variant of n most suited to current resource availability in the right location.
9: **function** CACHEUPDATE($n, (l, m)$ = **nil**) ▷ Insert n; variant m is already in
10: $e_d, ec_s, c_c \leftarrow$ CALCPERREQUESTBUDGETS(n)
11: $a_d, a_s, a_c \leftarrow$ CATALOGLOOKUPRES(n, e_d, ec_s, c_c, m)
12: $a^*, l^* \leftarrow \max_l a_l, \text{argmax}_l a_l$ ▷ Find optimal location and its accuracy
13: $v^* \leftarrow$ CATALOGLOOKUPACC(n, a^*) ▷ Look up variant with accuracy a^*
14: $m \leftarrow$ CACHEINSERT(l^*, v^*) **if** $m.v \neq v^*$ **or** $l \neq l^*$ **else** m
15: **return** m

▷ Calculate energy, energy/dollar and dollar budgets for executing model n on the mobile device, split between device/cloud and cloud only.
16: **function** CALCPERREQUESTBUDGETS(n)
17: $e, c \leftarrow$ REMAININGENERGY(), REMAININGCASH()
▷ 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. Δr_n is the cost to load n. Δt_n is the time since n was loaded, set to ∞ if n is not in any cache.
18: **def** RESPERREQ(r, l) = $(r - \Delta r_n T/\Delta t_n) f_n / (T \Sigma_{i \in \text{Cache}_l} f_i^2)$
19: $e_d, c_d \leftarrow$ RESPERREQ(e, "dev"), RESPERREQ(c, "$cloud$")
▷ Account for split models. t_n is the *fraction* of time spent executing the initial fragment of model n relative to executing the whole.
20: $e_s, c_s \leftarrow e_d t_n, c_d(1 - t_n)$
21: **return** $e_d, (e_s, c_s), c_d$

Continue of Algorithm 4.1

▷ Find the accuracies of the model variants for model n in the catalog that best match energy budget e, dollar budget c and split budget s. If the catalog lookup is due to a miss in the cache (i.e., m is nil, revert to a model that loads fast enough.

22: **function** CATALOGLOOKUPRES(n, e, s, c, m)

▷ CLX2A(n, r) returns accuracy of model n in location X using resources r

23: $a_e, a_s, a_c \leftarrow$ CLD2A(n, e), CLS2A(n, s), CLC2A(n, c)

▷ 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.

24: **if** m = **nil then**

25: $a_e, a_s, a_c \leftarrow \min(a_e^*, a_e), \min(a_s^*, a_s), \min(a_c^*, a_c)$

26: **return** a_e, a_s, a_c

27:

▷ Insert variant v in location l, where $l \in$ "dev", "$split$", "$cloud$"

28: **function** CACHEINSERT(l, v)

29: $s \leftarrow$ SIZE(v)

30: **if** $(a =$ CACHEAVAILABLESPACE$(l)) < s$ **then**

31: CACHEEVICT$(l, s - a)$ ▷ Reclaim space by evicting LRU models.

32: $m \leftarrow$ CACHELOAD(l, v) ▷ Load variant v to location l

33: **return** m

estimation works for the on-device energy budget (Line 19) (cloud cash budgets are identical, and device/cloud split budgets (Line 20) 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 $e_n = e f_n / T \Sigma_i f_i^2$, 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 $T f_i$ 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 18).

Given the estimated per-request resource budget for each location, we can consult the catalog to

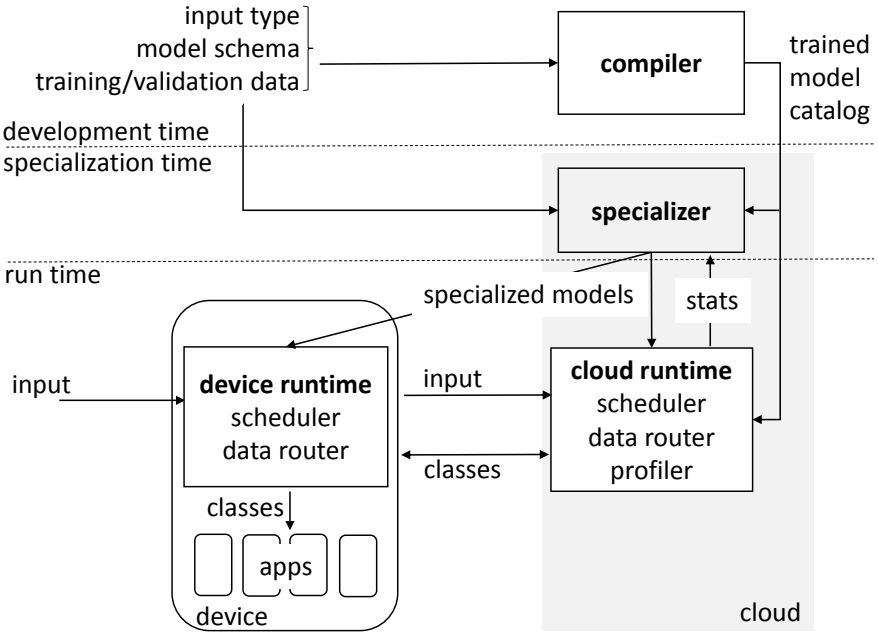


Figure 4.7: Architecture of the MCDNN system.

identify the variant providing the maximum accuracy for each location (Line 23) and update the cache at that location with that variant (Line 14). 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.

4.3.4 End-to-end architecture

Figure 4.7 illustrates the architecture of the MCDNN system. An application developer interested in using a DNN in resource-constrained settings provides the *compiler* with the type of input to which the model should be applied (e.g., faces), a model schema, and training data. The compiler derives a *catalog* of trained models from this data, mapping each trained variant of a model to its resource costs, accuracy, and information relevant to executing them (e.g., the runtime context in which they apply). When a user installs the associated application, the catalog is stored on disk on the device and

cloud and registered with the MCDNN *runtime* as handling input of a certain type for a certain app.

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

4.4 Evaluation

We have implemented the MCDNN system end-to-end. We adapt the open source Caffe [70] DNN library for our DNN infrastructure. As our mobile device, we target the NVIDIA Jetson board TK1 board [6], which includes the NVIDIA Tegra K1 mobile GPU (with roughly 300 gflops nominal peak performance), a quad-core ARM Cortex C15 CPU, and 2GB of shared memory between CPU and GPU. The Jetson is a developer-board variant of the NVIDIA Shield tablet [4], running Ubuntu 14.04 instead of Android as the latter does. For cloud server, we use a Dell PowerEdge T620 with an NVIDIA K20c GPU (with 5GB dedicated memory and a nominal peak of 3.52 tflops), a 24-core Intel Xeon E5-2670 processors with 32 GB of memory running Ubuntu 14.04. Where cloud costs are mentioned, we use Amazon AWS G2.2xlarge single-core GPU instances and c4.large CPU instances, with pricing data obtained in early September 2015. All energy measurements mentioned are directly measured unless otherwise specified, using a Jetson board and Shield tablet instrumented with a DC power analyzer [5].

Our main results are:

- Stand-alone optimizations yield 4-10× relative improvements in memory use of models with little loss of accuracy. In absolute terms, this allows multiple DNNs to fit within mobile/embedded device memory. However, the energy used by these models, though often lower by 2× or more,

is high enough that it is almost always more energy-efficient to offload execution when the device is connected to the cloud. Execution latencies on cloud CPU and GPU are also improved by a similar 2× factor, adequate to apply all but the largest models at 1 frame/minute at an annual budget of $10, but not enough to support a realistic 1 frame/second.

- Collective optimizations (specialization and sharing), when applicable, can yield 10× to 4 orders of magnitude reductions in memory consumption and 1.2× to over 100× reductions in execution speed and energy use. These improvements have significant qualitative implications. These models can fit in a fraction of a modern phone's memory making them suitable for both consumer mobile devices and for memory-constrained embedded devices such as smart cameras. It is often less expensive to execute them locally than to offload over LTE (or sometimes even WiFi). Finally, it is feasible to process 1 frame/second on a cloud-based CPU at $10/year.

- Both selecting which model to execute and where to run dynamically, as MCDNN does, can result in significant improvement in the number of requests served at fixed energy and dollar budget with little loss in accuracy. Essentially, common variations in classification task mix, optimization opportunities (such a specialization and sharing) and cloud-connectivity latencies defeat static model selection and placement schemes.

- MCDNN seems well-matched with emerging applications such as virtual assistants based on wearable video and query-able personal live video streams such as Meerkat[92] or Periscope[125]. We show promising early measurements supporting these two applications.

4.4.1 Evaluating optimizations

We first show the impact of *collective* optimizations, specialization and sharing. For specialization, we train and re-target progressively simpler models under increasingly favorable assumption of data skew, starting from assuming that 60% of all classes drawn belong to a group of at most size 21 (e.g., 60% of person sightings during a period all come from at most the same 21 people) to 90%/7 classes. We test these models on datasets with the same skew (e.g., 60% of faces selected from 21

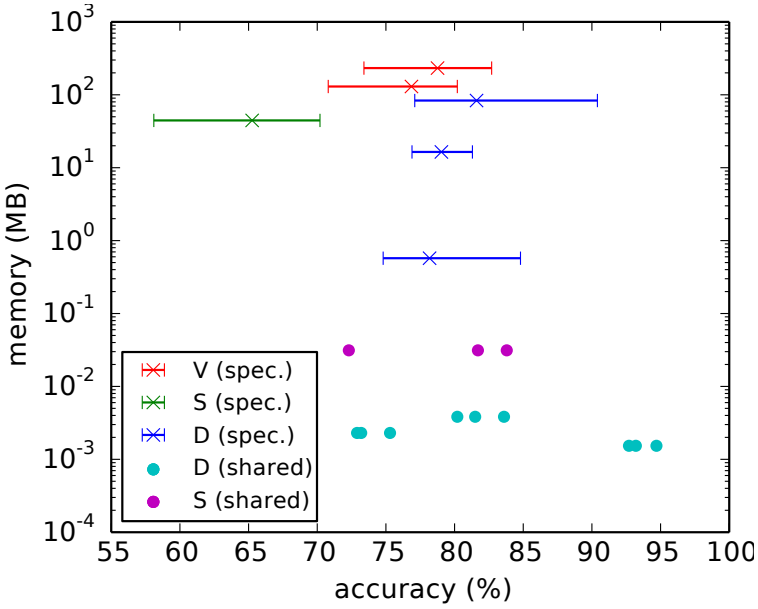


Figure 4.8: Impact of collective optimization on memory consumption

people, remaining selected randomly from 1000 people). The size, execution energy consumed, and latency of the model executed under these assumptions remains fixed, only the accuracy of using the model changes, resulting in horizontal line segments in the figure. We show mean, min and max under these assumptions. Two observations are key. First, **resource usage is dramatically lower** than even the statically optimized case. For instance (Figure 4.8; compare with Figure 4.3), specialized object recognition consumes 0.4-0.7mJ versus the stand-alone-optimized 7-16mJ. Second, **accuracy is significantly higher** than in the unspecialized setting. For instance the 0.4mJ-model has recognition rates of 70-82%, compared to the baseline 69% for the original VGGNet. Latency wins are similar as shown in Figure 4.10. These models take less energy to run on the device than to transmit to the cloud, and easily support 1 event *per second* on a $10/year GPU budget, and *often on a CPU budget as well*.

There is no free lunch here: specialization only works when the incoming data has class skew, i.e, when a few classes dominate over the period of use of the model. Class skew may seem an onerous restriction if the skew needs to last for hours or days. Table 4.2 details the time required to specialize a few representative variants of models including two models for face recognition and two for object

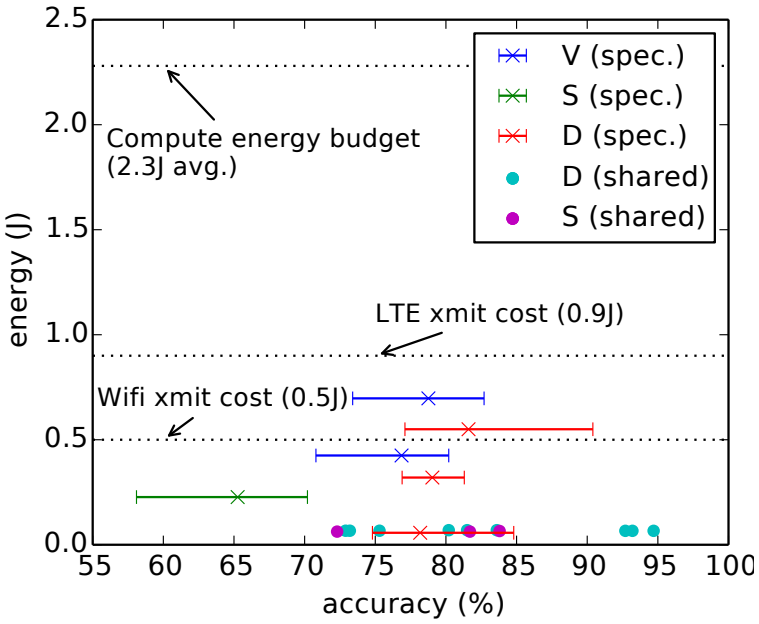


Figure 4.9: Impact of collective optimization on energy consumption

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

Table 4.2: Runtime overhead of specialization.

recognition (C4 is a smaller variant of C0 and A9 of A0; A0 is the standard AlexNet model for object recognition). Re-training these models from scratch takes hours to days. However, MCDNN's optimizations cut this overhead dramatically. If we only seek to re-target the model (i.e., only retrain the top layer), overhead falls to tens of seconds and pre-forwarding (see Section 5.2) training data through lower layers yields roughly 10-second specialization times. **MCDNN is thus well positioned to exploit data skew that lasts for as little as tens of seconds to minutes**, a capability that we believe dramatically broadens the applicability of specialization.

The benefits of sharing, when applicable, are even more striking. For scene recognition, assuming that the baseline scene recognition model (dataset S in Table 4.1) is running, we share all but its last

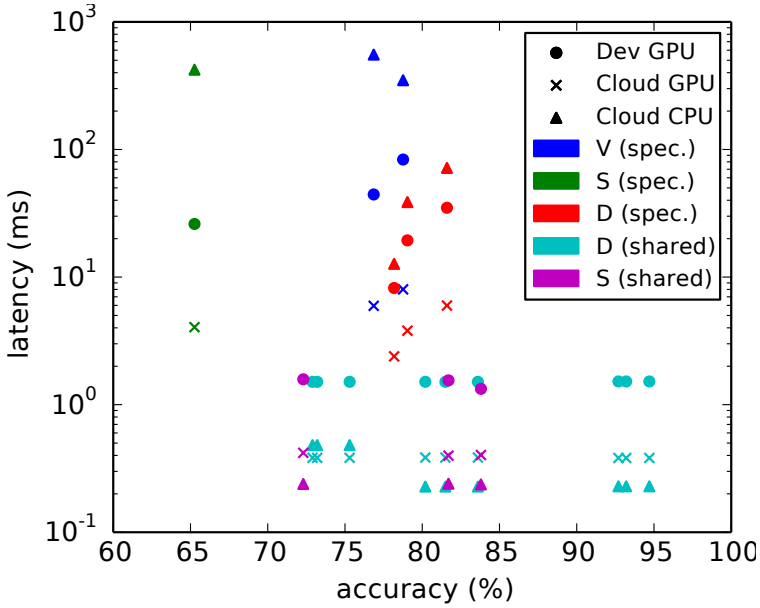


Figure 4.10: Impact of collective optimization on execution latency

layer to infer three attributes: we check if the scene is man-made or not (dataset M), whether lighting is natural or artificial (L), and whether the horizon is visible (H). Similarly, for face identification (D), we consider inferring three related attributes: age (Y), gender (G) and race (R). When sharing is feasible, the resources consumed by shared models are remarkably low: tens of kilobytes per model (cyan and purple dots in Figure 4.8), roughly 100mJ of energy consumption (Figure 4.9) and under 1ms of execution latency (Figure 4.10), representing over 100× savings in these parameters over standard models. Shared models can very easily run on mobile devices. Put another way, inferring attributes is "almost free" given the sharing optimization.

4.4.2 Evaluating the runtime

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

Selecting the right variant is especially important when executing on-device because the device has a finite-sized memory and paging models into and out of memory can be quite expensive in terms of power and latency as discussed in the previous section. MCDNN essentially pages in small

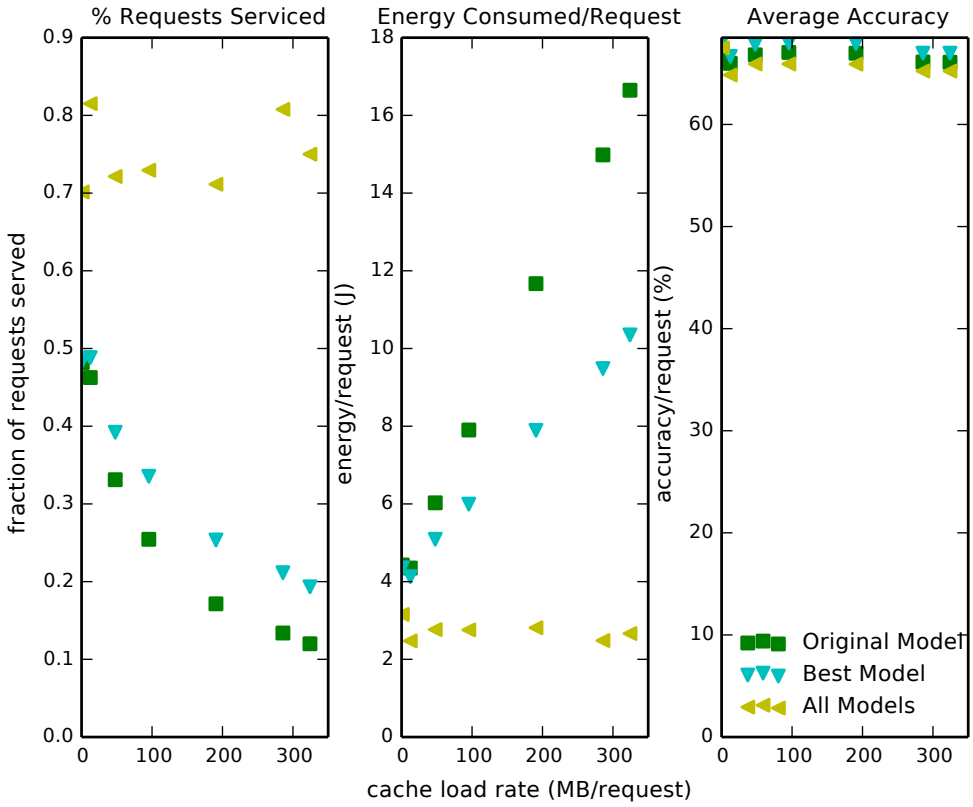


Figure 4.11: Impact of MCDNN's dynamically-sized caching scheme.

variants of models and increases their size over time as their relative execution frequency increases. To gauge the benefit of this scheme, in Figure 4.11, we use trace-driven evaluation to compare this dynamic scheme to two schemes that page *fixed-sized* models in LRU-fashion in and out of a 600MB-sized cache. In the "Original" scheme, the models used are the unoptimized models. In the "Best" scheme, we pick models from the knees of the curves in Figure 4.2, so as to get the best possible accuracy-to-size tradeoff. In the "All Models" scheme, the MCDNN model picks an appropriate model variant from a catalog of model sizes.

We generate traces of model requests that result in increasing *cache load rates* with respect to the original model: the load rate is the miss-rate weighted by the size of the missed model. We generate 10 traces per load rate. Each trace has roughly 36000 requests (one per second over 10 hours). Due to the energy required to service each request, none of the schemes is able to service all

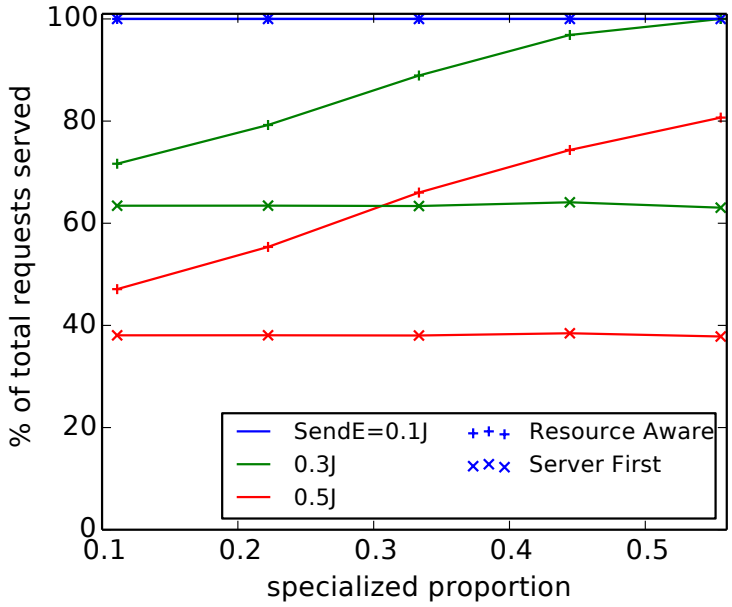


Figure 4.12: Impact of MCDNN's dynamic execution-location selection.

requests. The figure shows the number of requests served, the average energy expended per request, and the average accuracy per served request. The summary is that MCDNN constantly reduced the size of models in the cache so that even when the cache is increasingly oversubscribed with respect to fixed-size models, it still fits all the reduced-size models. As a result, MCDNN incurs minimal paging cost. **This results in a significantly higher percentage of requests served, lower average energy per request (in fact, the average energy remains fixed at execution cost for models rather than increasing amounts of paging costs), at a modest drop in accuracy**. Note that although the original and best models have higher accuracy *for requests that were served*, they serve far fewer requests.

We now move to the decision of where to execute. For each request, MCDNN uses dynamically estimated resource availability and system costs to decide where to execute. Given the relatively low cost of transmitting frames, a good strawman is a "server first" scheme that always executes on the cloud if cloud-budget and connectivity are available and on the device otherwise. This scheme if often quite good, but fails when transmission costs exceed local execution costs, which may happen

due to the availability of inexpensive specialized or shared models (Figure 4.9), an unexpectedly large client execution budget and/or high transmission energy costs (e.g., moving to a location far from the cloud and using a less energy-friendly transmission modality such as LTE). We examine this tradeoff in Figure 4.12.

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

4.4.3 MCDNN in applications

To understand the utility of MCDNN in practice, we built rough prototypes of two applications using it. The first, Glimpse, is a system for analyzing wearable camera input so as to provide personal assistance, essentially the scenario in Section 4.1. We connect the camera to an NVIDIA Shield tablet and thence to the cloud. In a production system, we assume a subsystem other than MCDNN (e.g., face detection hardware, or more generally a *gating* system [52]) that detects interesting frames and passes them on to MCDNN for classification. MCDNN then analyses these frames on the wearable and cloud. We currently use software detectors (which are themselves expensive) as a proxy for this detection sub-system. Figure 4.13 shows a trace reflecting MCDNN's behavior on face analysis (identification, age/race/gender recognition) and scene recognition tasks over a day[3]. The bottom panel shows the connectivity during the day. The straw man uses only the "best" models from knees of the curves in Figure 4.3 for all tasks. It always sends to cloud if possible (connected and has positive cost budget), and otherwise executes on device. We evaluate straw man with and without model

[3]The data fed to MCDNN in this experiment are synthesized based on statistics collected from the real world.

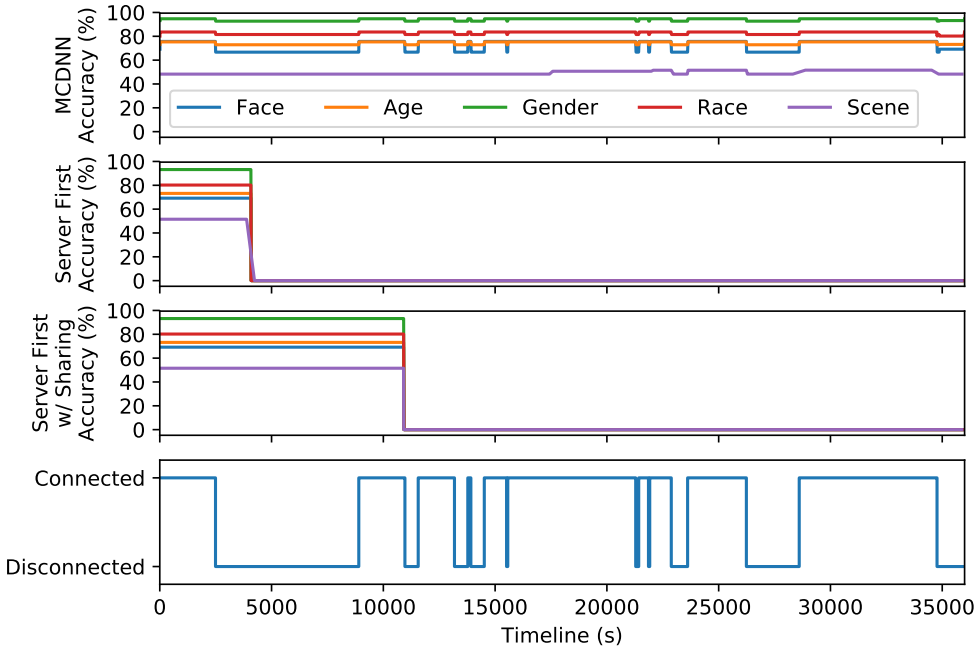


Figure 4.13: Accuracy of each application over time for the Glimpse usage. Each line shows a single application.

sharing for face analysis. Figure 4.13 (second and third row) shows that straw man can achieve good accuracy but exhausts the budgets quickly. Model sharing doubles the duration that applications last, but still fails to sustain for one day. MCDNN makes the trade-offs necessary to keep running through the day, and maintains a good accuracy over the time.

Our second application, Panorama, is a system that allows text querying of personal live video streams [92, 125] in real time. In this case, potentially millions of streams are sent to the cloud, and users can type textual queries (e.g., "man dog play") to discover streams of interest. The central challenge is to run VGGNet at 1-3 frames per second at minimal dollar cost. The key observation is that these streams have high class skew just as with wearable camera streams. We therefore aimed to apply specialized versions of VGGNet to a sample of ten 1-to-5-minute long personal video streams downloaded from YouTube; we labeled 1439 of 39012 frames with ground truth on objects present. VGGNet by itself takes an average of 364ms per frame to run on a K20 GPU at an average accuracy of 75%. MCDNN produces specialized variants of VGGNet for each video and reduces processing

overhead to 42ms at an accuracy of 83%. Panorama does not run on the end-user device: MCDNN is also useful in cloud-only execution scenarios.

4.5 Conclusions

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

Chapter 5

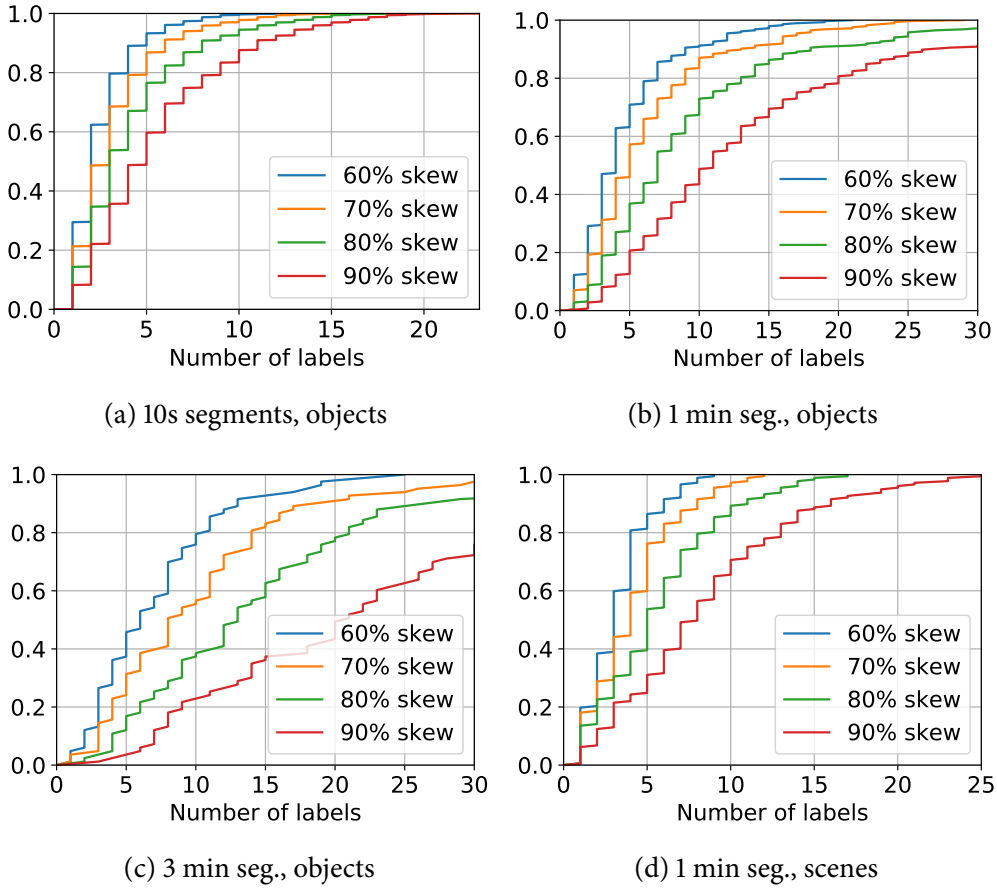
Sequential Specialization for Streaming Applications

Previous chapters describe two systems that optimize DNN inference under two serving scenarios. In this chapter, we introduce a general optimization called sequential specialization targeted for streaming applications. DNNs can classify across many classes and input distributions with high accuracy without retraining, but at relatively high cost. So applying them to classify video is costly. However, we show that in fact day-to-day video exhibits highly skewed class distributions over the short term, and that these distributions can be classified by much simpler models. We formulate the problem of detecting the short-term skews online and exploiting models based on it as a new sequential decision making problem dubbed the Online Bandit Problem, and present a new algorithm to solve it. When applied to recognizing faces in TV shows and movies, we realize end-to-end classification speedups of 2.4-7.8×/2.6-11.2× (on GPU/CPU) relative to a state-of-the-art convolutional neural network, at competitive accuracy.

5.1 Class skew in day-to-day video

Specialization depends on skew (or bias) in the temporal distribution of classes presented to the classifier. In this section, we analyze the skew in videos of day-to-day life culled from YouTube. We assembled a set of 30 videos of length 3 minutes to 20 minutes from five classes of daily activities: socializing, home repair, biking around urban areas, cooking, and home tours. We expect this kind of footage to come from a variety of sources such as movies, amateur productions of the kind that dominate YouTube and wearable videos.

We sample one in three frames uniformly from these videos and apply state-of-the-art face (derived from [100]), scene [143] and object recognizers [117] to every sampled frame. Note that these



(a) 10s segments, objects

(b) 1 min seg., objects

(c) 3 min seg., objects

(d) 1 min seg., scenes

Figure 5.1: Temporal skew of classes in day-to-day video

"oracle" recognizers can recognize up to 2622 faces, 205 scenes and 1000 objects respectively. For face recognition, we record the top-scoring label for each face detected, and for the others, we record only the top-scoring class on each frame. For object recognition in particular, this substantially undercounts objects in the scene; our count (and specialization) applies to applications that identify some distinctive subset of objects (e.g., all objects "handled" by a person). We seek to compare these numbers to the number of distinct recognized faces, scenes and objects that dominate "epochs" of τ = 10 seconds, 1 minute and 3 minutes.

Figure 5.1 shows the results for object recognition and scene recognition. We partition the sequence of frames into segments of length τ and show one plot per segment length. Each line in the plot corresponds to percentage skew $s \in \{60, 70, 80, 90\}$. Each line in the plots shows the cumulative

distribution representing the fraction of all segments where n labels comprised more than s percent of all labels in the segment. For instance, for 10-second segments (Figure 5.1(a)), typically roughly 100 frames, 5 objects comprised 90% of all objects in a segment 60% of the time (cyan line), whereas they comprise 60% of objects 90% of the time (dark blue).

In practice, detecting skews and training models to exploit them within 10 seconds is often challenging. As figures (b) and (c) show, the skew is less pronounced albeit still very significant for longer segments. For instance, in 90% of 3-minute segments, the top 15 objects comprise 90% of objects seen. The trend is similar with faces and scenes, with the skew significantly more pronounced, as is apparent from comparing figures (b) and (d); e.g. the cyan line in (d) dominates that in (b). We expect that if we ran a hand-detector and only recognized objects in the hand (analogously to recognizing detected faces), the skew would be much sharper.

Specialized models must exploit skews such as these to deliver appreciable speedups over the oracle. Typically, they should be generated in much less than a minute, handle varying amounts of skew gracefully, and deliver substantial speedups when inputs belong to subsets of 20 classes or fewer out of a possible several hundred in the oracle.

5.2 Specializing Models

In order to exploit skews in the input, we cascade the expensive but comprehensive *oracle model* with a (hopefully much) less expensive "compact" model. This *cascaded classifier* is designed so that if its input belongs to the frequent classes in the incoming distribution it will return early with the classification result of compact model, else it will invoke the oracle model. Thus if the skew dictates that n frequent classes, or *dominant classes*, comprise percentage p of the input, or *skew*, model execution will cost the overhead of just executing compact model roughly p% of the time, and the overhead of executing compact model and oracle sequentially the rest of the time. When p is large, the lower cost compact model will be incurred with high probability.

To be more concrete, we use state of the art convolutional neural networks (CNNs) for oracles. In particular, we use the GoogLeNet [121] as our oracle model, for object recognition; the VGG Net 16-layer version for scene recognition [143]; and the VGGFace network [100] for face recognition.

Task	Model	Accuracy (%)	FLOPs	CPU latency (ms)	GPU latency (ms)
Object (1000 classes)	GoogLeNet [121]	68.9	3.17G	779.3	11.0
	O1	48.9	0.82G	218.2 (×3.6)	4.4 (×2.5)
	O2	47.0	0.43G	109.1 (×7.1)	2.8 (×3.9)
Scene (205)	VGG [144]	58.1	30.9G	2570	28.8
	S1	48.9	0.55G	152.2 (×16.9)	3.36 (×8.6)
	S2	40.8	0.43G	141.5 (×18.2)	2.44 (×11.8)
Face (2622)	VGG-Face [100]	95.8	30.9G	2576	28.8
	F1	84.8	0.60G	90.1 (×28.6)	2.48 (×11.6)
	F2	80.9	0.13G	40.4 (×63.7)	1.93 (×14.9)

Table 5.1: Oracle classifiers versus compact classifiers in top-1 accuracy, number of FLOPs, and execution time. Execution time is feedforward time of a single image without batching on Caffe [70], a Linux server with a 24-core Intel Xeon E5-2620 and an NVIDIA K20c GPU.

The compact models are also CNNs. For these, we use architectures derived from the corresponding oracles by systematically (but manually) removing layers, decreasing kernel sizes, increasing kernel strides, and reducing the size of fully-connected layers. The end results are architectures (O[1—2] for objects, S[1—2] for scenes and F[1—2] for faces) that use noticeably less resources (Table 5.1), but also yield significantly lower average accuracy when trained and validated on unskewed data, i.e., the same training and validation sets for oracle models. For instance, O1 requires roughly 4× fewer FLOPs to execute than VGGFace, but achieves roughly 70% of its accuracy. Detailed model architecture definition of compact models can be found in Appendix B.

However, in our approach, we train these compact models to classify *skewed* distributions observed during execution, denoted by *specialized classifier*, and their performance on skewed distributions is the critical measure. In particular, to generate a specialized model, we create a new training dataset with the data from the n dominant classes of the original data, and a randomly chosen subset from the remaining classes with label "other" such that the dominant classes comprise p percent of the new data set. We train the compact architecture with this new dataset.

Figure 5.2 shows how compact models trained on skewed data and cascaded with their oracles

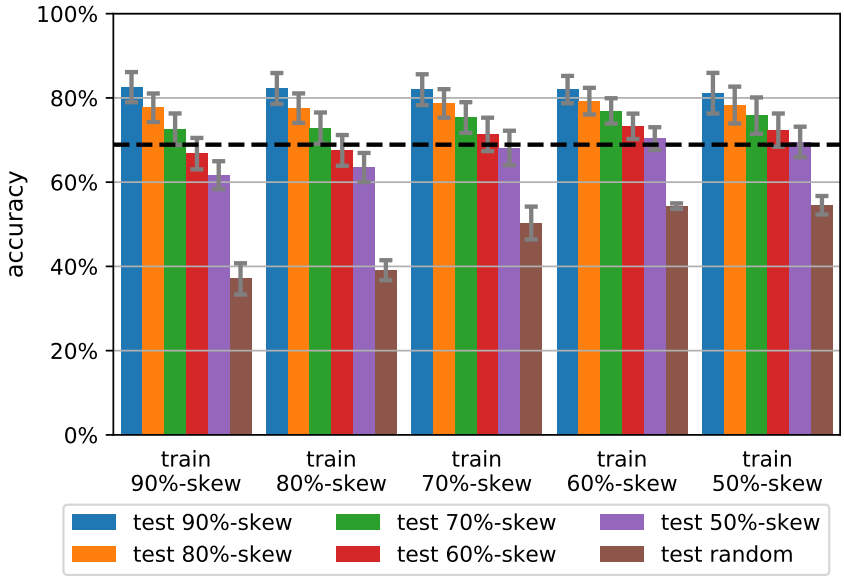


Figure 5.2: When compact model O1 is trained by various skews and cascaded with the oracle, accuracy of the cascaded classifier tested by various skews for 10 dominant classes

perform on validation data of different skews. We first analyze the case where n = 10, for various combinations of training and validation skews for model O1. Recall from Table 5.1 that O1 delivers only 70% of its accuracy on unskewed inputs. However, when training and testing is on skewed inputs, the numbers are much more favorable. When O1 is trained on p=90% skewed data with n=10 dominant classes, it delivers over 84% accuracy on average (the left-most dark-blue bar). This is significantly *higher* than the oracle's average of 68.9% (top-1 accuracy), denoted by the horizontal black line. We also observed from Figure 5.2 that when O1 is trained on 60% skewed data, the cascaded classifier maintains high accuracy across a wide range of testing skews from 90% to 50%. Therefore, in what follows, we use 60% skew as *fixed training skew* to specialize object compact models in the rest of paper (similarly 70% fixed skew for scene and 50% for face). Figure 5.3 shows that, where n is varied for O1, the cascaded classifier degrades very gracefully with n. Finally, Figure 5.4, which reports similar measurements on compact models S[1—2] and F[1—2] shows that these trends carry over to scene and face recognition.

Finally, we note that since skews are only evident at test-time, specialized models must be trained extremely fast (ideally a few seconds at most). We use two techniques to accomplish this. First, before

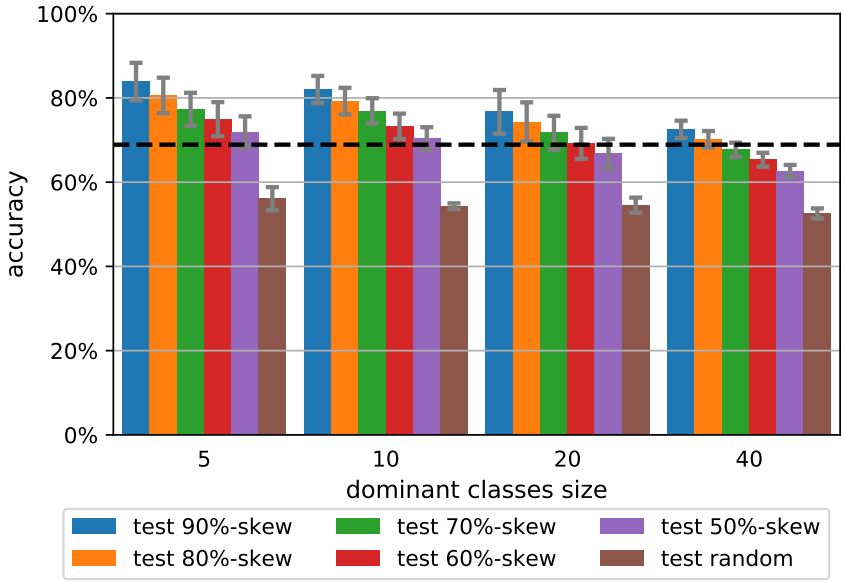


Figure 5.3: Accuracy of O1 trained by 60% skew and tested by various skews for different number of dominant classes. The dashed line shows the accuracy of GoogLeNet as a baseline. All experiments repeated 5x with randomly selected dominant sets.

we begin processing any inputs, we train all model architectures on the full, unskewed datasets of their oracles. At test time, when the skew n, p and the identity of dominant classes is available, we only retrain the top (fully connected and softmax) layers of the compact model. The lower layers, being "feature calculation" layers do not need to change with skew. Second, as a pre-processing step, we run all inputs in the training dataset through the lower feature-calculation layers, so that when re-training the top layers at test time, we can avoid doing so. This combination of techniques allows us to re-train the specialized model in roughly 4s for F1 and F2 and 14s for O1/O2, many orders of magnitude faster than fully re-training these models.

5.3 Sequential Model Specialization

5.3.1 The Oracle Bandit Problem (OBP)

Let $x_1, x_2, \ldots, x_i, \ldots \in X = \mathbb{R}^n$ be a stream of images to be classified. Let $y_1, y_2, \ldots, y_i, \ldots \in Y = [1, \ldots, k]$ be the corresponding classification results. Let $\pi : \mathbb{I}^+ \to \mathbb{I}^+$ be a partition over the stream. Associate the distribution T_j with partition j, so that each pair (x_i, y_i) is sampled from $T_{\pi(i)}$. Intu-

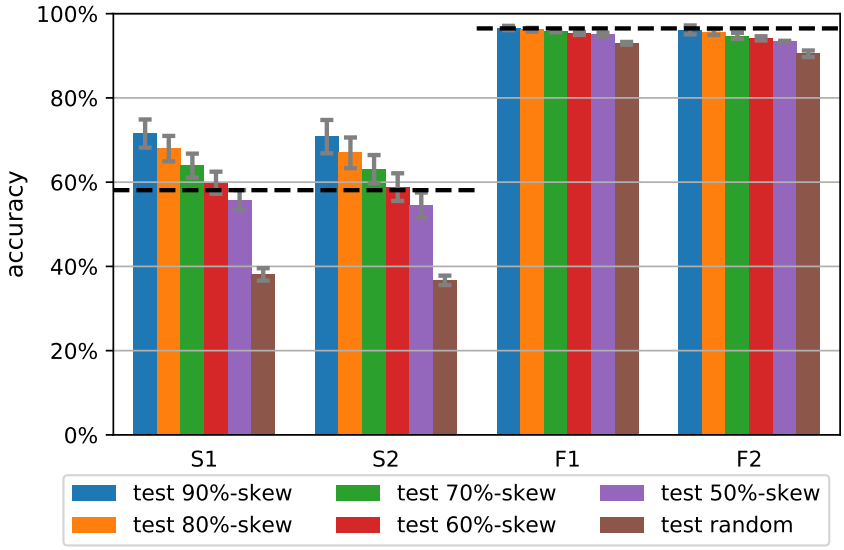


Figure 5.4: Accuracy of scene classifiers trained by 70% fixed skew and 10 dominant classes and face classifiers trained by 50% fixed skew and 10 dominant classes. Dashed lines show the accuracy of the oracle classifier for scene and face recognition task.

itively, π partitions, or segments, $\ldots, x_i, \ldots$ into a sequence of "epochs", where elements from each epoch j are drawn independently from the corresponding stationary distribution T_j. Thus, for the overall series, samples are drawn from an abruptly-changing, piece-wise stationary distribution. At test time, neither results y_i nor partitions π are known.

Let $h^* : X \to Y$ be a classifier, designated the "oracle" classifier, trained on distribution T^*. Intuitively T^* is a mixture of all distributions comprising the oracle's input stream: $T^* = \sum_j T_j$. Let R^* be the cost (e.g., number cycles), assumed invariant across X, needed to execute h^* on any $x \in X$. At test time, on each input x_i, we can always consult h^* at cost R^* to get a label y_i with some (high) accuracy a^*.

Let $m_1, \ldots, m_M$ be *model architectures*, such as those of O1, O2, S1, S2, F1 and F2 in Table 5.1. Suppose each architecture m_k is trained *offline* on T^* to obtain a "template" classifier h_k. We assume that re-targeting template h_k to a new size-j set of dominant classes has a flat cost R_0.

Finally, for each set of dominant classes D, the corresponding *specialized classifier* h_D is trained by re-targeting some template h_k, using a dataset that draws half its examples from classes in D and the rest (with a single label "other") from $Y - D$. Let the cost of executing h_d be R_{h_d}. Chaining h_D

with h^* gives a *cascaded* classifier $\hat{h}_D(x) \triangleq$ if $y = h_D(x) \in D$, return y, otherwise return $h^*(x)$. Note that executing $\hat{h}_D$ will either cost R_{h_d} (in the case that the condition is true), or $R_{h_d} + R^*$ in the case that it is false. Given that $R_{h_d} \ll R^*$, developing and using specialized classifiers h_D can thus reduce costs significantly. Since x_i is drawn from some distribution T, each classifier $\hat{h}_D$ also has *cost* that belongs to a corresponding distribution, which we write as $R_{T\hat{h}_D}(x_i)$.

Now consider a policy (or algorithm) P that, for each incoming image x_i belonging to stationary distribution $T_{\pi(i)}$ as above, selects a classifier $\hat{h}_D^{(i)}$ (for some set choice of D), and applies it to x_i. The classifier selected could also include the oracle. The expected total cost of this policy, $R_p = |\cup_i \{\hat{h}_D^{(i)}\}|R_0 + \Sigma_i E_{x_i \sim T_{\pi(i)}}(R_{T_{\pi(i)}\hat{h}_D^{(i)}}(x_i))$. We seek a minimal-cost policy: $P^* = \arg\min_P R_P$ that maintains average accuracy within a threshold τ_a of oracle accuracy a^*.

5.3.2 The Windowed ε-Greedy (WEG) Algorithm

A close look at the policy cost above provides some useful intuition on what good policies should do. First, given the high fixed cost R_0 of re-targeting models as opposed to just running them, re-targeting should be infrequent. We expect re-targeting to occur roughly once an epoch. Second, the cost of running the cascade is much lower than that of running the oracle *if the input x_i is in the dominant class set D* and higher otherwise. It is important therefore to identify promptly when a dominant set D exists, produce a specialized model h_D that does not lose too much accuracy, and revert back to the oracle model when the underlying distribution changes and D is no longer dominant. We provide a heuristic exploration-exploitation based algorithm (Algorithm 5.1) based on these intuitions.

The algorithm runs in three phases.

1. **Window Initialization** [lines 2 - 7] identifies the dominant classes of the current epoch. To do so, we run the oracle on a fixed number w_{min} (= 30 in our implementation) of examples. The DomClasses helper identifies the dominant classes in the window as those that appear at least k (=2 to 3) times in the window. Appendix C provides a simple analysis of how we determine the threshold k for a given window size. If the dominant classes are each within τ_r (= 2) of those of the previous epoch, we conclude the previous epoch is continuing and fold

Algorithm 5.1 Windowed ε-Greedy (WEG)

1: $j, S_0 \leftarrow 1, [\,]$
▷ Note: τ_r, τ_a, τ_{FP} and ε below are hyper-parameters.
Window Initialization Phase
2: Repeat w_{min} times
3: $y_t \leftarrow h^*(x_t)$
4: $S_j \leftarrow S_j \oplus [y_t]$ ▷ Append new sample
5: **if** $||\textsc{DomClasses}(S_{j-1}), \textsc{DomClasses}(S_j)|| \leq \tau_r$ **then**
▷ dominant classes match sufficiently, old epoch continues
6: $S_j \leftarrow S_{j-1} \oplus S_j$
7: $w \leftarrow |S_j|$ and go to Line 8

Template Selection Phase
8: $D \leftarrow \textsc{DomClasses}(\text{last } w \text{ elements in } S_j)$
9: Estimate acc. $a_{\hat{h}_D}$ of $\hat{h}_D$; use p^* derived from S_j (Equation 5.1)
10: **if** $a_{\hat{h}_D} \geq a^* + \tau_a$ **then**
11: train specialized classifier h_D on dominant classes D
12: go to Line 16 ▷ Exploit cascaded classifier $\hat{h}_D$
13: $y_t \leftarrow h^*(x_t)$ ▷ Else, continue exploring with oracle
14: $S_j \leftarrow S_j \oplus [y_t]$
15: go to Line 8

Specialized Classification Phase
16: $n_c, n^*, S \leftarrow 0, 0, S_j$
17: $y_t, c \leftarrow \hat{h}_D(x_t)$ ▷ exploit; $c = 0|1$ if—if-not cascaded to oracle
18: $n^* \leftarrow (c \textbf{ or } \text{rand}() \geq \varepsilon)\ ?\ n^* : n^* + (h^*(x_t) \neq y_t)$
19: $n_c \leftarrow n_c + c$ ▷ Increment if $\hat{h}_D$ did not use oracle
20: Estimate acc. $a_{\hat{h}_D}$ of $\hat{h}_D$; use p^* derived from S_j (Equation 5.1)
21: **if** $a_{\hat{h}_D} < a^* + \tau_a$ **or** $\frac{n^*}{n_c \cdot \varepsilon} > \tau_{FP}$ **then**
▷ Exit specialized classification
22: $j \leftarrow j + 1$ ▷ Potentially start new epoch j
23: go to Line 2 ▷ Go back to check if distribution has changed
24: **else**
25: $S \leftarrow S \oplus [y_t]$; go to Line 17

information collected on it into that for the current epoch S_j.

2. **Template Selection** [lines 8 - 15] Given a candidate set of dominant classes D, we estimate (Equation 5.1 below details precisely how) the accuracy of the cascaded classifier $\hat{h}_D$ for various template classifiers h_i when specialized to D and their current empirical probability skew p^* derived from measured data S_j. Estimating these costs instead of explicitly training the corresponding specialized classifiers h_D is significantly cheaper. If the estimate is within a threshold τ_a (= 0.05 for object and scene recognition, and -0.05 for face recognition since the accuracy of oracle is higher) of the oracle, we produce the specialized model and go to the specialized classification phase. If not, we continue running the oracle on inputs and collecting more information on the incoming class skew.

3. **Specialized Classification** [lines 16 - 17] The specialized classification phase simply applies the current cascaded model $\hat{h}_D$ to inputs (Line 17) until it determines that the distribution it was trained on (as represented by D) does not adequately match the actual current distribution. This determination is non-trivial because in the specialization phase, we wish to avoid consulting the oracle in order to reduce costs. However, the oracle is (assumed to be) the only unbiased source of samples from the actual current distribution.
   We therefore run the oracle in addition to the cascaded model with probability ε (= 0.01), as per the standard ε-greedy policy for multi-arm bandits. Given the resulting empirical estimate p^* of skew, we can again estimate the accuracy of the current cascade $\hat{h}_D$ as per Equation 5.1. If the estimated accuracy of the cascade is too low, or if the classification results of $\hat{h}_D$ cascade are different from the oracle too often (we use a threshold τ_{FP} = 0.5), we assume that the underlying distribution may have shifted and return to the Window Initialization phase.

Finally, we focus on estimating the expected accuracy of the cascaded classifier given the current skew p of its inputs (i.e., the fraction of its inputs that belong to the dominant class set). The accuracy

of cascaded classifier $\hat{h}_D$ can be estimated by:

$$a_{\hat{h}_D} = p \cdot a_{in} + p \cdot e_{in \to out} \cdot a^* + (1 - p) \cdot a_{out} \cdot a^* \tag{5.1}$$

where a_{in} is the accuracy of specialized classifier h_D on n dominant classes, $e_{in \to out}$ is the fraction of dominant inputs that h_D classifies as non-dominant ones, and a_{out} is the fraction of non-dominant inputs that h_D classifies as non-dominant (note that these inputs will be cascaded to the oracle). We have observed previously (Section 5.2) that these parameters a_{in}, $e_{in \to out}$, a_{out} of specialized classifier h_D are mainly affected only by the size of the dominant class D, not the identity of elements in it. Thus, we pre-compute these parameters for a fixed set of values of n (averaging over 10 samples of D for each n), and use linear interpolation for other ns at test time.

5.4 Evaluation

We implemented the WEG algorithm with a classification runtime based on Caffe [70]. The system can be fed with videos to produce classification results by recognizing frames. Our goal was to measure both how well the large specialized model speedups of Table 5.1 translated to speedups in diverse settings and on long, real videos. Further we wished to characterize the extent to which elements of our design contributed to these speedups.

5.4.1 Synthetic experiments

First, we evaluate our system with synthetically generate data in order to study diverse settings. For this experiment, we generate a time-series of images picked from standard large validation sets of CNNs we use. Each test set comprises of one or two segments where a segment is defined by the number of dominant classes, the skew, and the duration in minutes. For each segment, we assume that images appear at a fixed interval (1/6 seconds) and that each image is picked from the testing set based on the skew of the segment. For an example of a segment with 5 dominant classes and 90% skew, we pre-select 5 classes as dominant classes and pick an image with 90% probability from the dominant classes and an image with 10% probability from the other classes at each time of image

Segments	Object				Scene			
	disabled		enabled		disabled		enabled	
	acc(%)	lat(ms)	acc(%)	lat(ms)	acc(%)	lat(ms)	acc(%)	lat(ms)
(n=5,p=.8)	69.5	11.6	77.0	6.0	57.6	28.9	65.2	12.0
(n=10,p=.8)	66.7	11.7	72.5	7.4	57.2	28.9	57.8	18.6
(n=10,p=.9)	71.8	11.6	78.0	5.9	59.1	28.8	63.5	15.4
(n=15,p=.9)	68.7	11.6	68.9	9.1	57.8	28.8	57.2	22.6
(random)	68.1	12.1	68.1	11.5	59.1	28.9	59.1	28.8
(n=10,p=.9) +(random)	67.9	11.8	70.2	9.1	57.0	28.8	56.0	22.6
(n=15,p=.9) +(n=5,p=.8)	70.6	11.6	73.9	7.8	61.1	28.7	63.0	17.1

Segments	Face			
	disabled		enabled	
	acc(%)	lat(ms)	acc(%)	lat(ms)
(n=5,p=.8)	95.2	28.7	95.1	9.2
(n=5,p=.9)	97.0	28.6	96.2	6.7
(n=10,p=.9)	95.4	28.5	94.3	11.0
(random)	95.9	28.5	95.9	28.5
(n=5,p=.9) +(random)	96.2	28.5	96.2	17.6
(n=10,p=.9) +(n=10,p=.8)	95.8	28.5	95.2	10.4

Table 5.2: Average accuracy and GPU latency of recognition using WEG algorithm over segments of synthetic traces . For the segment column, each parenthesis indicates a segment of 5 minutes with the number of dominant classes and the skew.

arrival over 5 minutes duration. Images in a class are picked in a uniform random way. We also generate traces with two consecutive segments with different configurations to study the effect of moving from one context to the other.

Table 5.2 shows the average top-1 accuracies and per-image processing latencies using GPU for the recognition tasks with and without the specializer enabled. The results are averaged over 5 iterations for each experiment. The specializer was configured to use the compact classifiers O2 for objects, S2 for scenes, and F2 for face recognition from Table 5.1.

The following points are worth noting. (i) (Row 1 and it's sub-rows) *WEG is able to detect and exploit skews over 5-minute intervals and get significant speedups over the oracle while preserving accuracy.* For the single segment cases, the GPU latency speedup per-image was 1.3× to 2.0×, 1.3× to 2.4×, and 2.6× to 4.3×, for object, scene, and face, respectively. However, due to WEG's overhead these numbers are noticeably lower than the raw speedups of specialized models (Table 5.1). When the number of dominant classes increase, the specializer latency increases because it alternates between exploration and exploitation to recognizes more dominant classes. The latency also increases when the skew of dominant classes decreases because specializer cascades more times to oracle model when using the cascaded classifier. (ii) (Row 2) *WEG is quite stable in handling random inputs*, essentially resorting to the oracle so that accuracy and latency are unchanged. (iii) (Rows 3 and 4) *WEG is able to detect abrupt input distribution changes* as the accuracy remains comparable to oracle accuracy, but with significant speedups when the distribution is skewed (Row 4).

To further understand the limit of the WEG algorithm, we studied how frequently class distributions can change before our technique stops showing benefit. We evaluated face recognition with synthetic traces, changing the distribution every 10/20/30 sec. The WEG algorithm then yields speedups of 0.95/1.19/1.48× with roughly unchanged accuracy. For face recognition therefore, WEG stops gaining benefit for distributions that lasts less than 20 secs.

5.4.2 Video experiments

We now turn to evaluating WEG on real videos. However, we were unable to find a suitable existing dataset to show off specialization. We need several minutes or more of video that contains small

video	length (min)	oracle acc(%)	oracle CPU lat	oracle GPU lat	WEG acc(%)	WEG CPU lat	WEG GPU lat
Friends	24	93.2	2576	28.97	93.5	538(×**4.8**)	7.0(×**4.1**)
Good Will Hunting	14	97.6	2576	28.84	95.1	231(×**11.2**)	3.7(×**7.8**)
Ellen Show	11	98.6	2576	29.26	94.6	325(×**7.9**)	4.7(×**6.2**)
The Departed	9	93.9	2576	29.18	93.5	508(×**5.1**)	6.9(×**4.2**)
Ocean's Eleven / Twelve	6	97.9	2576	28.97	96.0	1009(×**2.6**)	12.3(×**2.4**)

Table 5.3: Accuracy and average processing latency per frame on videos with oracle vs. WEG (latencies are shown in ms).

video	special rate(%)	cascade rate(%)	trans. special	dom. size	window size
Friends	88.0	7.5	51	2.8	41.8
Good Will Hunting	95.9	3.4	4	3.5	37.5
Ellen Show	93.7	4.8	19	1.7	47.4
The Departed	92.0	10.3	9	2.4	40.0
Ocean's Eleven / Twelve	80.1	18.0	23	2.0	52.2

Table 5.4: Key statistics from WEG usage. (a) "special rate" is the percentage of time that specializer uses the cascaded classifier; (b) "cascade rate" is the percentage of time that a cascaded classifier cascades to oracle classifier; (c) "trans. special" is the number of transitions to specialized classification phase; (d) "dom. size" is the average dominant classes size; (e) "window size" is the average window size.

subsets of (the oracle model's) classes that may change over time. In videos of real-world activity, this happens naturally; in popular benchmarks, not so much. For example, the videos in YouTube Faces [129] are short (average 6 sec, max 200 sec) and typically only contain one person. Similarly, clips in UCF-101 [118] have mean length of 7.2 sec (max 71 sec) and focus on classifying *actions* for which no oracle model exists. Finally, the sports 1M dataset [73] assigns labels *per video* instead of *per frame*.

As a consequence, we hand-labeled video clips from three movies, one TV show, and an interview and manually labeled the faces in the videos [1]. The names of video clips with lengths are listed in

[1]The dataset is released at `http://syslab.cs.washington.edu/research/neurosys`.

Table 5.3. Note that we used the entire videos for Friends and Ellen Show, while we used a video chunk for the movies. For these experiments, we used F2 as the compact classifier.

Table 5.3 shows the average accuracies and average latencies for processing a frame for 5 videos. We generated these by first extracting all faces from the videos to disk using the Viola Jones detector. We then ran WEG on these faces and measured the total execution time. Dividing by the number of faces gave the average numbers shown here. The most important point is that *even on real-world videos, WEG is able to achieve very significant speedups over the oracle*, ranging from 2.6×-11.2× (CPU) and 2.4×-7.8× (GPU).

To understand the speedup, we summarize the statistics of WEG execution in Table 5.4. "Special rate" indicates the percentage of time that specializer exploits cascaded classifier to reduce the latency, while "cascade rate" reveals the percentage of time that a cascaded classifier cascades to the oracle classifier, thus hurting performance. Higher special rate and lower cascade rate yield more speedup. The cascade rate of "Ocean's Eleven" is significantly higher than that of other videos. We investigated this and found that the specialized compact CNN repeatedly made mistakes on one person in the video, which led to a high cascade rate. "Trans. special" counts the number of times WEG needed to switch between specialized and unspecialized classification to handle the distribution changes and insufficient exploration. The average dominant classes sizes ("dom. size") show that the real videos are skewed to fewer dominant classes than the configurations used in the synthetic experiments. This explains why our system achieved higher speedup on real videos than on synthetic data. Overall, the statistics show that *the dataset exercise WEG features such as skew estimation, cascading and specialization*.

To understand better the utility of WEG's features, we performed an ablation study: (a) We disable the adaptive window exploration (Line 5-7 in Algorithm 5.1), and use a fixed window size of 30 and 60. (b) We use the skew of dominant classes in the input distribution as the training skew for specializing compact CNNs instead of using the fixed (50%) training skew suggested in Section 5.2. (c) We apply a simple (but natural) criterion to exit from the specialized classification phase: WEG now exits when the current skew is lower than the skew when it entered into specialized classification phase instead of using the estimated accuracy as soft indicator.

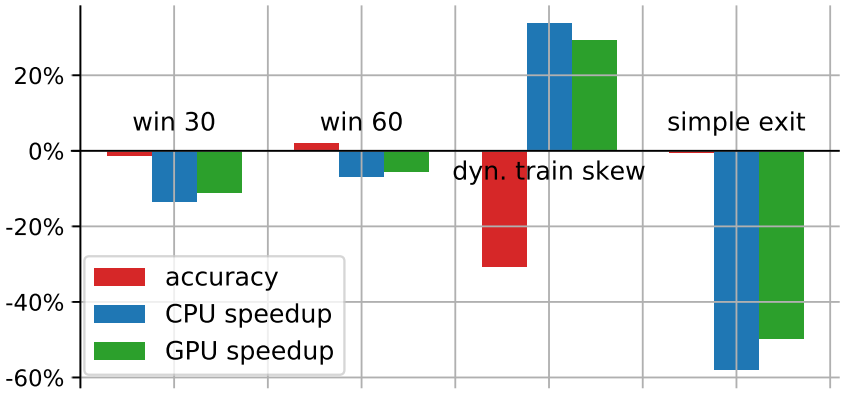


Figure 5.5: Change in accuracy (absolute difference) and speedup (relative) when individual features are disabled.

Figure 5.5 shows the comparison between these variants and WEG algorithm in accuracy and CPU / GPU speedups when recognizing faces on Friends video. In the figure we show the absolute differences in accuracy and relative differences in CPU / GPU speedup. (a) Fixed window size (30 and 60) variants achieve similar accuracy but lower speedup. As table 5.4 ("window size" column) shows, the adaptively estimated size for the window is between 30 and 60. In general, too small a window fails to capture the full dominant classes, yielding specializers that exit prematurely. Too large a window requires more work by the oracle to fill up the window. (b) Using variable rather than fixed skew for training achieves more speedup, but suffers from 30% loss in accuracy. This is because the training skew is usually very high. As discussed in Section 5.2, training on highly skewed data produces models vulnerable to false positives in "other" classes. (c) The simple exit variant achieves almost comparable accuracy while the latency is more than 50% higher than our system. It demonstrates the value of our accuracy estimate in modeling the accuracy of cascaded classifiers and to prevent premature exit from the specialized classification phase. *In summary, the key design elements of WEG each have a role in producing fast and accurate results.*

5.5 Conclusion

We characterize class skew in day-to-day video and show that the distribution of classes is often strongly skewed toward a small number of classes that may vary over the life of the video. We

further show that skewed distributions are well classified by much simpler (and faster) convolutional neural networks than the large "oracle" models necessary for classifying uniform distributions over many classes. This suggests the possibility of detecting skews at runtime and exploiting them using dynamically trained models. We formulate this sequential model selection problem as the Oracle Bandit Problem and provide a heuristic exploration/exploitation based algorithm, Windowed ε-Greedy (WEG). Our solution speeds up face recognition on TV episodes and movies by 2.4-7.8× on a GPU (2.6-11.2× on a CPU) with little loss in accuracy relative to a modern convolutional neural network.

Chapter 6

Conclusion

As deep learning revolutionizes many areas in machine learning, DNN-based applications are being applied to broader real-world scenarios and are becoming an important workload for both cloud and edge computing. However, it is challenging to design a serving system that satisfies various constraints such as latency, energy, and cost, while achieving high efficiency.

This dissertation identified three major challenges: scheduling and resource allocation, resource management, and model selection. To address these challenges, we investigated the problem through a series of thorough studies of DNN models and operators, accelerator performance profiles, accuracy and resource trade-off of various model optimization methods, and distribution patterns of DNN workload. Based on these, I explored the designs of DNN serving system ranging from a cloud serving system, a mobile-cloud execution framework, to a general optimization framework for streaming applications. For the cloud environment, I presented Nexus that addresses the resource allocation and scheduling problem and distributes the workload such that it achieves high utilization on accelerators while meeting the latency SLOs. For the mobile-cloud execution, I designed MCDNN that carefully manages the limited resource budgets and systematically selects optimized models generated by various approximation techniques in order to optimize the accuracy and performance. Third, for a streaming workload, I devised sequential specialization, a general optimization that generates specialized models based on current workload distribution and switches between specialized classification mode and oracle classification mode appropriately.

Evaluation of the three systems on realistic workloads demonstrates that we can improve the throughput and reduce the cost and energy consumption significantly by better scheduling, resource management, and model selection algorithms, while meeting all of the relevant constraints. The

work also points to avenues for further improvement and extension of serving systems that can be applied to other types of neural networks, applications, and platforms.

Discussion and Future Work

Recurrent neural networks (RNNs) In this dissertation, we mostly focus on convolutional neural networks. However, recurrent neural networks are also an important category and widely used in natural language processing tasks. In general, principles used in this dissertation can be applied to RNN models as well, but require additional attention. First, one primary distinction between RNNs and CNNs is that the input to a RNN model is a sequence of data with uncertain length. It implies that the workload provided to RNNs can vary request by request. This introduces new challenges to scheduling and load balancing due to the higher variance in execution costs. Second, some RNN models like language models require significant amounts of memory in order to include more words and symbols. It is possible that the main memory of a single GPU is not large enough to fit one model. In this case, it is necessary to split a model to multiple GPUs and even multiple servers. Serving systems need to carefully optimize the overhead from synchronization and communication.

Generalization to other applications The systems described in this dissertation can serve not only DNN applications but also other similar applications. For example, video analytics applications, which doesn't necessarily require DNNs, can still gain higher efficiency on accelerators when batching requests. Therefore, the Nexus system, which optimizes for batched execution, can also handle such applications and improve cluster utilization.

New hardware This dissertation mostly explores server- and mobile- GPUs as accelerators. However, there is an increasing trend of customized FPGAs and ASICs [71, 29, 107, 1, 33, 34] to accelerate DNN execution. Certain principles such as batching still apply to these new hardware for better efficiency. But it raises a bigger challenge regarding how to generate highly efficient kernels for each new hardware. Recently, new compiler frameworks such as TVM [23], Glow [112], DLVM [128] aim to provide high-level programming frameworks that enable developers to easily write and tune highly efficient kernels on new hardware. In the future, we should integrate such compiler frameworks into the serving system so that it can select the best kernel implementation for each hardware or even use

just-in-time (JIT) compilation to generate a new kernel based on currently available resources.

Integration with other systems In certain settings, DNNs are not used in standalone applications, but as an intermediate process in a larger system. For example, Optasia [90] treats DNNs as user defined functions (UDFs) in the SQL query. MLlib [93] generalizes MapReduce framework and Spark [140] to support machine learning tasks. Such a system can delegate its DNN execution to Nexus for better efficiency. It would be worth exploring as to whether Nexus is general and expressive enough to not only develop and deploy new applications but also serve as a subsidiary system to such execution engines.

Bibliography

[1] Arm machine learning processor. https://developer.arm.com/products/processors/machine-learning/arm-ml-processor.

[2] Tesla data center gpus for servers. https://www.nvidia.com/en-us/data-center/tesla/.

[3] Twitch. https://www.twitch.tv/.

[4] Gpu-based deep learning inference: A performance and power analysis. https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf, 2015.

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

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

[7] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[8] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[10] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Fast cholesky factorization on gpus for batch and native modes in magma. *Journal of Computational Science*, 20:85–93, 2017.

[11] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *OSDI*, pages 739–753, 2016.

[12] ARM. Mali graphics processing. `https://www.arm.com/products/graphics-and-multimedia/mali-gpu`.

[13] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3), May 2002.

[14] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

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

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

[17] Denis Baylor, Eric Breck, and Heng-Tze Cheng. Tfx: A tensorflow-based production-scale machine learning platform, 2017.

[18] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[19] Korte Bernhard and J Vygen. *Bin-Packing*, pages 449–465. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

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

[21] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

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

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[24] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier 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.

[25] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of The 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.

[26] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2285–2294, 2015.

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

[28] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[29] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[30] John Cocke. Global common subexpression elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, pages 20–24. ACM, 1970.

[31] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[32] Eduardo Cuervo, Aruna Balasubramanian, Dae ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[33] Ian Cutress. Cambricon, makers of huawei's kirin npu ip, build a big ai chip and pcie card. `https://www.anandtech.com/show/12815/cambricon-makers-of-huaweis-kirin-npu-ip-build-a-big-ai-chip-and-pcie-card`.

[34] Ian Cutress. Huawei shows unannounced kirin 970 at ifa 2017: Dedicated neural processing unit. `https://www.anandtech.com/show/11804/`

huawei-shows-unannounced-kirin-970-at-ifa-2017-dedicated-neural-processing-unit, 2017.

[35] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the Twenty-sixth Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[37] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[38] Chaitanya Desai and Deva Ramanan. Detecting actions, poses, and objects with relational phraselets. In *Proceedings of the 12th European Conference on Computer Vision - Volume Part IV*, ECCV'12, 2012.

[39] Docker. Docker swarm. `https://github.com/docker/swarm`.

[40] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.

[41] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655, 2014.

[42] Kathryn A. Dowsland. Simulated annealing. In Colin R. Reeves, editor, *Modern heuristic techniques for combinatorial problems*, pages 20–69. Mc Graw-Hill, 1993.

[43] Facebook. Caffe2. `https://caffe2.ai/`.

[44] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[46] Aurélien Garivier and Eric Moulines. On upper-confidence bound policies for non-stationary bandit problems. In *Proceedings of the 22nd International Conference on Algorithmic Learning Theory (ALT)*, 2011.

[47] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[48] Google. Cloud automl vision. `https://cloud.google.com/vision/automl/docs/`.

[49] Google. Kubernetes: Production-grade container orchestration. `https://kubernetes.io/`.

[50] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[51] Marty Hall and J Paul McNamee. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest*, 18(2):255, 1997.

[52] Seungyeop Han, Rajalakshmi Nandakumar, Matthai Philipose, Arvind Krishnamurthy, and David Wetherall. GlimpseData: Towards continuous vision-based personal analytics. In *Proceedings of the 1st Workshop on Physical Analytics (WPA)*, 2014.

[53] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016.

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

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

[56] S. Hare, A. Saffari, and P. H. S. Torr. Struck: Structured output tracking with kernels. In *2011 International Conference on Computer Vision*, 2011.

[57] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. ACM, 2015.

[58] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–238. ACM, 2015.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[60] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

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

[62] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 269–286, 2018.

[63] Junxian Huang, Feng Qian, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g LTE networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 225–238, 2012.

[64] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016.

[65] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2018.

[66] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[67] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

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

[69] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[70] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[71] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[72] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[73] Andrej Karpathy et al. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[74] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional networks for fast, low power mobile applications. *ICLR*, 2016.

[75] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

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

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

[78] B. Kveton and M. Valko. Learning from a single labeled face and a stream of unlabeled data. In *Automatic Face and Gesture Recognition (FG), 2013 10th IEEE International Conference and Workshops on*, 2013.

[79] T.L Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Adv. Appl. Math.*, 6(1), March 1985.

[80] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[81] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. Pretzel: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626. USENIX Association, 2018.

[82] Yunseong Lee, Alberto Scolari, Matteo Interlandi, WA Redmond, Markus Weimer, and Byung-Gon Chun. Towards high-performance prediction serving systems. In *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[83] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *Proceedings of the 14th Annual Conference of the International Speech Communication Association (Interspeech)*, 2013.

[84] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. A convolutional neural network cascade for face detection. In *CVPR*, 2015.

[85] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. A convolutional neural network cascade for face detection. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, 2015.

[86] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 1, page 3, 2014.

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

[88] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Diannaoyu: An instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[89] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[90] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 57–70. ACM, 2016.

[91] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in c++. *ACM SIGPLAN Notices*, 33(8):17–22, 1998.

[92] Meerkat. Meerkatstreams website. `http://meerkatstreams.com/`, 2015.

[93] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[94] Microsoft. Custom vision. `https://azure.microsoft.com/en-us/services/cognitive-services/custom-vision-service/`.

[95] Microsoft. Virtual machine scale sets. `https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-overview`.

[96] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[97] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

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

[99] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[100] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. In *BMVC*, 2015.

[101] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. Deep face recognition. In *BMVC*, volume 1, page 6, 2015.

[102] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[103] H. Pirsiavash and D. Ramanan. Detecting activities of daily living in first-person camera views. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012.

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

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

[106] Swati Rallapalli, Aishwarya Ganesan, Krishna Chintalapudi, Venkat N Padmanabhan, and Lili Qiu. Enabling physical analytics in retail stores using smart glasses. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 115–126, 2014.

[107] Naveen Rao. Intel nervana neural network processors (nnp) redefine ai silicon. `https://ai.intel.com/intel-nervana-neural-network-processors-nnp-redefine-ai-silicon/`.

[108] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[109] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[110] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[111] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[112] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[113] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. Rio: a system solution for sharing I/O between mobile systems. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 259–272, 2014.

[114] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[115] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[116] Tom Simonite. Apple's 'neural engine' infuses the iphone with ai smarts. `https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/`, 2017.

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

[118] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[119] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[120] R. Sutton and A. Barto. *Reinforcement Learning, an introduction*. MIT Press/Bradford Books, 1998.

[121] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[122] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[123] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

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

[125] Twitter. Twitter periscope website. `http://www.periscope.tv`, 2015.

[126] Michal Valko, Branislav Kveton, Ling Huang, and Daniel Ting. Online semi-supervised learning on quantized graphs. In *Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, pages 606–614, Corvallis, Oregon, 2010. AUAI Press.

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

[128] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[129] Lior Wolf, Tal Hassner, and Itay Maoz. Face recognition in unconstrained videos with matched background similarity. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 529–534. IEEE, 2011.

[130] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. A scalable approach to activity recognition based on object use. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

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

[132] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[133] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[134] Zhixiang Xu, Matt Kusner, Minmin Chen, and Kilian Q. Weinberger. Cost-sensitive tree of classifiers. In *ICML*, 2013.

[135] Linjie Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. A large-scale car dataset for fine-grained categorization and verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3973–3981, 2015.

[136] Qiang Yang, Charles X. Ling, Xiaoyong Chai, and Rong Pan. Test-cost sensitive classification on data with missing values. *IEEE Trans. Knowl. Data Eng.*, 18(5):626–638, 2006.

[137] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[138] Dong Yu, Jinyu Li, Dong Yu, Mike Seltzer, and Yifan Gong. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2014.

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

[140] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[141] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[142] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.

[143] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Proceedings of the Twenty-eighth Annual Conference on Neural Information Processing Systems (NIPS)*, 2014.

[144] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.

[145] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[146] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

Appendix A

Hardness of Fixed-rate GPU Scheduling Problem (FGSP)

We now justify the use of an approximate algorithm for GPU cluster scheduling. We define the *Fixed-rate GPU Scheduling Problem* (FGSP), which is a highly restricted version of the general problem, and we show that even the restricted version is intractable.

FGSP:

Input - models $M_i, 1 \leq i \leq n$ with corresponding latencies L_i, latency bounds B_i and GPU count C. (The latencies correspond to the fixed rates.)

Output - Partition of the models into C sets so that in each set S we have $D + L_i \leq B_i, \forall i \in S$ where $D = \sum_{i \in S} L_i$ is the duty cycle for the set.

We show that FGSP is strongly NP-hard by reduction from 3-PARTITION [45].

Theorem 1. *FGSP is strongly NP-complete.*

Proof. We start with a given instance of 3-PARTITION which consists of a bound B and $3n$ $\frac{B}{4} \leq a_1, a_2 \ldots, a_{3n} \leq \frac{B}{2}$; the goal of 3-PARTITION is to partition the a_is into triples such that the sum of each triple is B. Observe that wlog we may assume that $\sum_{1 \leq i \leq 3n} a_i = nB$.

From the given instance of 3-PARTITION we create an instance of FGSP by setting $L_i = 2B + a_i, B_i = 9B + a_i, \forall 1 \leq i \leq 3n, C = n$.

It is clear that if there exists a solution to the 3-PARTITION instance then the same partition into n triples yields a partition of the FGSP instance into $C = n$ sets so that $D + L_i \leq 9B + a_i$ since $D = 7B$ and $L_i = 2B + a_i$. In the other direction suppose there exists a solution to FGSP. Observe that in any solution to FGSP every set can have at most 3 models because otherwise the duty cycle D would exceed $8B$ and then the constraint $D + L_i \leq B_i$ would be violated for any i in the set, since

$D + L_i > 10B$ but $B_i < 10B$. Since there are a total of $3n$ models and $C = n$ sets every set must have exactly 3 models, i.e. every set must be a triple. Since $D + L_i \leq B_i$ for any i in the set, we have that $D + 2B + a_i \leq 9B + a_i$ or $D \leq 7B$. But this implies that in every triple the sum of the L_is is at most $7B$ or the sum of the corresponding a_is is at most B. But since the sum of all the n triples is nB and each triple is at most B it must be that the sum of each triple is exactly B. This means that the partition of models of the FGSP instance into sets is also a solution for the partition of the corresponding a_i into triples in 3-PARTITION.

□

Appendix B
Compact Model Architectures

Table B.1 shows the six compact models used in the Chapter 5. All convolution and fully-connected layers are followed by a ReLU activation layer. We created these models by systematically applying the following operations to publicly available model architecture, such as AlexNet [77] and VGGNet [117]: (a) reduce the number of feature maps, or increase stride size in a convolution layer; (b) reduce the size of a fully-connected layer; (c) merge two convolution layers or a convolution layer and a max-pooling layer into a single layer. The models are unremarkable except that they have fewer operations and layers than original versions and hence run faster, yet achieve high accuracy *when applied to small subsets of the original model's domain*. In fact, how these architectures are derived is orthogonal to the WEG algorithm (Algorithm 5.1). We have tested two fully automatic approximation techniques, tensor factorization [74] and representation quantization [64] to generate compact models as well.

O1	O2	S1	S2
input 3 × 227 × 227		input 3 × 227 × 227	
conv[11, 96, 4, 0]	conv[11, 64, 4, 0]	conv[11, 96, 4, 0]	conv[11, 64, 4, 0]
pool[3, 2]		pool[3, 2]	
conv[5, 256, 2, 2]		conv[5, 256, 2, 2]	
pool[3, 2]		pool[3, 2]	
conv[3, 384, 1, 1]	conv[3, 256, 1, 1]	conv[3, 384, 1, 1]	conv[3, 256, 1, 1]
conv[3, 384, 1, 1]	conv[3, 256, 1, 1]	conv[3, 384, 1, 1]	conv[3, 256, 1, 1]
conv[3, 256, 1, 1]		conv[3, 256, 1, 1]	
pool[3, 2]		pool[3, 2]	
fc[4096]	fc[1024]	fc[2048]	fc[1024]
fc[4096]	fc[2048]	fc[2048]	fc[2048]
fc[1000]		fc[205]	

F1	F2
input 3 × 152 × 152	
conv[3, 64, 2, 1]	
pool[2, 2]	
conv[3, 128, 1, 1]	conv[3, 128, 2, 1]
pool[2, 2]	
conv[3, 256, 1, 1]	conv[3, 128, 2, 1]
pool[2, 2]	
conv[3, 256, 1, 1]	conv[3, 256, 2, 1]
pool[2, 2]	
fc[2048]	fc[1024]
fc[2048]	fc[1024]
fc[2622]	

Table B.1: Architecture of compact models O1, O2, S1, S2, F1, and F2. The table specifies convolution layers as [kernel size, number of feature maps, stride, padding]; max-pooling layers as [kernel size, stride]; fully-connected layers as [output size].

Appendix C

Determine Dominant classes

In the WEG algorithm, an important component is to decide the dominant classes from a sliding window (function DomClasses in Algorithm 5.1). The decision used in the algorithm is fairly simple: return the classes as dominant classes that appear at least k times in a sliding window w of classification result history from the oracle h^*, where k is the minimum support number.

Here we use a simple model to analyze how to choose the minimum support number k for different window sizes w in the WEG algorithm. Suppose N is the number of total classes classified by the oracle h^* and the accuracy of h^* is a^*. If the classification result from the oracle is wrong, the probability that the oracle classifies the input to each of the other classes is assumed to be equivalent. Suppose the input sequences are drawn independently from a skewed distribution T which has n dominant classes with skew p. Denote the set of dominant classes as $\mathcal{D}$ and set of non-dominant set as $\mathcal{O}$. Based on these definitions, we can compute the probability of a single class that the oracle classifier h^* outputs given the input sequences. Consider one dominant class $\ell \in \mathcal{D}$, the probability that it is output by the oracle is:

$$\text{prob}(\ell \in \mathcal{D}) = \frac{1}{n} \cdot \hat{p} = \frac{1}{n}\left(p \cdot a^* + p(1-a^*)\frac{n-1}{N-1} + (1-p)(1-a^*)\frac{n}{N-1}\right) \tag{C.1}$$

And the probability of a non-dominant class $\ell' \in \mathcal{O}$ that is output by the oracle is:

$$\text{prob}(\ell' \in \mathcal{O}) = \frac{1}{N-n}\left((1-p)a^* + (1-p)(1-a^*)\frac{N-n-1}{N-1} + p(1-a^*)\frac{N-n}{N-1}\right) \tag{C.2}$$

From Equation C.1 and C.2, we can then derive the probability of a dominant class or a non-

index	N	a^*	n	p	w	k	p_{in}	p_{out}
1	1000	0.68	5	0.9	30	2	0.896	6.52E-5
2	1000	0.68	10	0.9	30	2	0.558	6.53E-5
3	1000	0.68	10	0.9	60	2	0.891	2.64E-4
4	1000	0.68	10	0.7	60	2	0.789	4.80E-4
5	1000	0.68	10	0.7	90	2	0.933	1.08E-3
6	205	0.58	10	0.9	90	2	0.959	0.019
7	205	0.58	10	0.9	90	3	0.872	1.31E-3
8	205	0.58	0	N/A	90	3	N/A	9.29E-3

Table C.1: Probability p_{in} and p_{out} under various N, a^*, n, p and w, k. In row 8, n = 0 and p = N/A indicates that the distribution has no skew and is uniform across all N classes.

dominant class that is classified at least k times in the window w by using binomial distribution. Intuitively, we hope the probability of a dominant class that appear at least k times (p_{in}) in the window be as high as possible, while the probability of any non-dominant class (p_{out}) be as low as possible. We considered different combinations of N, a^*, n, p under different w and k settings, and computed p_{in} and p_{out}. Table C.1 shows how p_{in} and p_{out} changes under different settings. From the table, we can tell that with an increase in the number of dominant classes and a decrease in skew, we need to increase the size of window to have a higher probability that the dominant classes can be detected in the window. However, when the window size is larger, we also need to increase the minimum support number k (Row 6 and 7) to limit the probability that a non-dominant class appears k times. In addition, when there is no skew in the distribution (Row 8), the minimum support number k is also effective at filtering out most of the non-dominant classes. In practice, we set $k = 2$ for $w < 90$ and $k = 3$ for $w \geq 90$.