Improving Deep Learning Training and Inference with Dynamic Hyperparameter Optimization

Angela Jiang

CMU-CS-20-112

May 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Gregory R. Ganger, Chair
David G. Andersen
Michael Kaminsky, BrdgAI
Michael Kozuch, Intel Labs
Padmanabhan S Pillai, Intel Labs
Rahul Sukthankar, Google

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by a gift from Nvidia Corporation and a gift from Intel Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: machine learning, deep learning, computer vision, edge computing, hyperparameters

To my mom, Yonglan Wang,
for always being behind me

Abstract

Over the past decade, deep learning has demonstrated state-of-the-art accuracy on challenges posed by computer vision and natural language processing, revolutionizing these fields in the process. Deep learning models are now a fundamental building block for applications such as autonomous driving, medical imaging, and neural machine translation. However, many challenges remain when deploying these models in production. Researchers and practitioners must address a diversity of questions, including how to efficiently design, train, and deploy resource-intensive deep learning models and how to automate these approaches while ensuring robustness to changing conditions.

This dissertation provides and evaluates new ways to improve the efficiency of deep learning training and inference, as well as the underlying systems' robustness to changes in the environment. We address these issues by focusing on the many hyperparameters that are tuned to optimize the model's accuracy and resource usage. These hyperparameters include the choice of model architecture, the training dataset, the optimization algorithm, the hyperparameters of the optimization algorithm (e.g., the learning rate and momentum) and the training time budget. Currently, in practice, almost all hyperparameters are tuned once before training and held static thereafter. This is suboptimal as the conditions that dictate the best hyperparameter value change over time (e.g., as training progresses or when hardware used for inference is replaced). We apply dynamic tuning to hyperparameters that have traditionally been considered static. Using three case studies, we show that using runtime information to dynamically adapt hyperparameters that are traditionally static can increase the efficiency of machine learning training and inference.

First, we propose and analyze Selective-Backprop, a new importance sampling approach that prioritizes examples with high loss in an online fashion. In Selective-Backprop, the examples considered challenging is a tunable hyperparameter. By prioritizing these challenging examples, Selective-Backprop trains to a given target error rate up to 3.5x faster than static approaches.

Next, we explore AdaptSB, a variant of Selective-Backprop that dynamically adapts how we prioritize challenging examples. In Selective-Backprop, the priority assigned to examples of differing degrees of difficulty is held static. In AdaptSB, we treat the priority assigned to different classes of examples as a tunable hyperparameter. By dynamically tailoring example prioritization to the dataset and stage in training, AdaptSB outperforms Selective-Backprop on datasets with label error.

Finally, we propose and analyze Mainstream, a video analysis system that adapts concurrent applications sharing fixed edge resources to maximize aggregate result quality. In Mainstream, we consider the degree of application sharing to be a tunable parameter. Mainstream automatically determines at deployment time the right trade-off between using more specialized DNNs to improve per-frame accuracy and keeping more of an unspecialized base model. We show that Mainstream improves mean event detection F1-scores by up to 87x, compared to static approaches.

Acknowledgments

There's many people I need to recognize, whose support and guidance made this thesis possible. In doing so, I'll undoubtedly miss some names; so to those that I miss, I apologize.

First, thank you to my advisor, Greg Ganger for his unwavering support throughout this journey. Greg always made me feel comfortable presenting crazy ideas, whether it be the hypothetical intros I wrote while searching for my thesis topic, or the announcement that I might leave the program to join a presidential campaign. Thank you guiding me away from the bad ideas, nurturing the good ones, all while making sure I stayed encouraged.

I am grateful for Dave Andersen, who became my defacto co-advisor by the end. One of my favorite memories with Dave is walking around Google Brain, meeting people and asking for swag from Diana. Thank you Dave, for always going the extra mile for me, whether that was in securing me an NVIDIA TitanX, or a TensorFlow t-shirt.

I also want to thank my other thesis committee members, Michael Kaminsky, Michael Kozuch, Babu Pillai, and Rahul Sukthankar. Iterating on SysML intros together for hours was one of my most memorable times with Michael Kaminsky. Thanks Michael, for always holding me to a high bar. I've been collaborating with Michael Kozuch since my first year. I'll always remember his support in the last 24 hours of our 2018 ATC deadline, where he caught a bug just before I left for the airport, and was up until 3 a.m. the next day helping me perfect the wording of a footnote. Thanks Michael, for always being willing to dig one level deeper. Babu was one of the first collaborators I talked with about pursuing something like Mainstream. He taught me about edge computing on the fourth floor in CIC and since that day, he's been with me for every deadline until the end. I'm extremely fortunate to have had Rahul join my thesis committee. Thank you Rahul, for your insights and thoroughness as we worked on the thesis together.

I'm also grateful to other mentors who helped me along the way, including Zachary Lipton, Ishai Menache, and Fabian Bustamante. Zack and I met at UCSD visit days and quickly hit it off, making him one of my first friends in the field. Thank you for also becoming one of my most dedicated collaborators, and for teaching me about machine learning, writing, and jazz. Thanks Ishai, for your infectious positivity and encouragement in both research and in fun. I'll always remember deciding last-minute to ditch OSDI talks and sprinting for the boat to take us into downtown Savannah. Thank you Fabian, for investing in me when I was first starting out in the field. It feels like just yesterday you were coaching me for my SRC talk in the Starbucks of the Palmer House.

Thanks to Deb Cavlovich, Joan Digney, and Karen Lindenfelser for making the administrative aspects of the PhD program easy. Thanks Angy, for being College Mom, and the other thing too.

FAWN group was more than a research lab. Thanks Dong Zhou, Anuj Kalia, Huanchen Zhang, Conglong Li, Sol Boucher, Thomas Kim, Chris Canel, Giulio Zhou, and Daniel Wong for the fun times, advice, and for motivating me to go to the gym. One of the most rewarding parts of the PhD is that I found a lifelong friend in Chris. The courage, morals, and high standards that Chris displays every day pushes me to be a better person and friend. I know that I could tell Chris

tomorrow that I want to run for president, and he'd not only believe in me but he'd also find a way to help. Thomas, please keep in contact with me after I leave CMU! Life is so much more fun with you around. Daniel's uncanny ability to avert disasters that I never saw coming made him an invaluable collaborator. His empathy, kindness, and wisdom makes him an invaluable friend. I discovered Anuj's humor, kindness, and frankness over long walks all over the country. Over time, I got to see another side, that once Anuj sets his mind to something he'll transcend what you thought was humanly possible. Thank you Anuj, for continuing to surprise me every day.

To my Pittsburgh friends, thanks for making the PhD one long party, with the occasional breaks for research. Thanks Ellen, for rooming with me our first year so that we could kick the party off together. Thanks as well for being part of the coolest crew from Columbia, so you could introduce me to Rohan and Zan. Thanks Goran, Ellen, and Rohan, for the poker nights and giving me the chance to earn the great distinction of wiping Noam out.[1] Thanks Nic, Nick, Mariah, Sol, and Fait for the 4 a.m. dance parties, I'm still waiting to have another. Thanks Nika, for giving the best advice. Thanks Erik, for always teaching me something new. Thanks Mariah, for all the brunches, I'll meet you in the promised land of Seattle. Thanks Fait, for the real talks we have every time we manage to see each other. Thanks Nikos, for your friendship which has endured three moves in three years. Thanks Zheru, for always keeping it real, meaningful, and fun. Thanks to David and Naama, who have the uncanny ability to always be traveling, yet always be available for friends who need it. Thanks to John, for setting the gold standard of friendship and for reminding me to be as cool as an eggplant. Thanks as well to Akanksha Tyagi, Anish Sevekari, Colin White, Daniel Anderson, Deby Katz, Greg Kehne, Isaac Grosof, Rajesh Jayaram, Roie Levin, Ryan Kavanagh, and many many others.

I'm indebted to my friends outside of CMU. Early in life, I won the best friend lottery by meeting Kate. Thank you Kate, for over ten years of being a supportive, hilarious, non-judgmental, and insightful friend. And thanks for bringing Lucas into my life, who is the second funniest person I've ever met. (I have to save first place just in case someone funnier comes along.) Paige and I are basically the same person, so I obviously think she's one of the coolest people in the world. Thanks for always validating and supporting my inner diva and for putting up with my flaky texting habits. Meeting Hank was a transformational experience for me. Thank you Hank, for teaching me to trust myself, to dream big, and to do good. This thesis and what's to come will always be a reflection of our time together.

Finally, thank you to my family, who offers me unconditional love and support. Dad, thank you for being a role model in work ethic, leadership, and friendship. Bo bo, thank you for caring for me in times of need. Thanks to Meng Meng for your generosity and enthusiasm. Hayden, thanks for being the calmest one in the family and probably the funniest too.[2] Liz is always six years ahead of me and it shows. Thank you Liz, for paving the path and for watching out for me as I navigate my way through. Anything I achieve in life is a testament to my mom's ability to empower those around her. Thank you Mom, for dedicating your fearlessness of life towards providing Liz and I the best.

[1] We were playing Short Deck for the first time, not Texas Hold'em.

[2] Liz is also strongly under consideration.

Contents

Chapter 1

Introduction

In the past decade, there has been a resurgence of interest in the use of deep neural networks (DNNs) for emerging and sometimes critical applications like autonomous driving, medical imaging, and neural machine translation. Such networks often have hundreds of layers and millions of parameters, and are only possible due to the vastly greater computational resources available today. To adequately train such large networks without over-fitting requires a vast amount of labeled data, made possible through developments in big data techniques and crowd sourcing efforts. Despite the computational and training challenges, the payoff is significant for applications. For example, DNNs have achieved human-level or better accuracy in some image recognition tasks [41, 115], and they vastly outperform most other recognition approaches in computer vision.

Despite the astonishing progress made in machine learning and its supporting infrastructure, there are still many obstacles to designing and deploying DNNs for real applications. While initial research focused primarily on advancing the state-of-the-art accuracy achieved by deep learning, researchers and practitioners are now tackling an array of issues arising from the end-to-end use of a DNN. These issues include reducing the cost of training and inference, and increasing the interpretability, fairness and security of deployed networks. This dissertation provides and evaluates new ways to improve the efficiency and robustness of DNN training and inference, discussed next.

1.1 Efficiency and robustness for DNNs in production

Training Efficiency DNNs are much more computationally demanding than traditional machine learning (ML) approaches, making DNN training time-consuming and financially expensive. A single model that performs neural translation between English and German took Google engineers 250,000 GPU hours to train, about a $200,000 price on Google Compute Engine [11]. Training BERT, a state-of-the-art language model, took four days using four TPU pods running

16 TPU chips each [25]. Dettmers et al. [24] estimate that training would take 10–17 days on 8 GPUs. These prices only capture a fraction of the resources needed to prepare a DNN because training is not a one time cost. Training a model requires iterations from debugging, tuning of hyperparameters and retraining with new labeled data. Hence, training models of production scale is prohibitively expensive for many practitioners.

Inference Efficiency Deep learning models are larger than traditional ML models, thus requiring more computational resources to run. State-of-the-art object detection networks, such as Faster-RCNN, operate as slowly as 1 frame per second (FPS) on a modern GPU [52]. Inference efficiency is often a first-class design goal of production models as DNNs are often used for low-latency tasks, requiring sub-millisecond response times for some applications like autonomous driving. Inference often runs in edge computing further tightens the latency budget available for inference by reducing the network latency between end users and the inference servers. In addition, serving inference requests from edge datacenters pushes the limits of inference hardware. This is because although datacenters can use tens or hundreds of GPUs or even TPUs for inference, edge deployments often have only a few servers, a single mini-PC, or an embedded device.

Robustness Machine learning practitioners optimize DNN models and systems infrastructure to take into account a variety of application-specific conditions. These include assumptions about the distribution of the training dataset, deployment conditions, and application service level objectives (SLOs). Incorrect assumptions can lead to higher training cost and suboptimal models and applications. For example, a model designed to run in a datacenter may have too large a memory requirement for a low-power edge device. Importantly, optimizing once for application-specific conditions may not be adequate. In a production environment, conditions are constantly in flux. Incoming data changes over time (e.g., seasonal shifts exhibited in a traffic camera); but upstream applications require different latency SLOs, and deployment hardware changes. ML systems should allow for adaptation of application-specific optimizations in order to provide robustness to real-world changes.

1.2 The case for dynamic hyperparameter optimization

Machine learning practitioners must tune a variety of hyperparameters to improve the task's accuracy and to reduce resources used during training. Examples include the choice of DNN model architecture, the training dataset, the optimization algorithm, the hyperparameters of the optimization algorithm (e.g., the learning rate and momentum) and the training time budget. Currently, in practice, almost all hyperparameters are static. They are tuned once during training, for instance using a grid search or a Bayesian hyperparameter optimizer. Such a static approach has low robustness to changing conditions, such as changes in the dataset or the available computational resources. In this thesis, we show that, to improve the efficiency of ML processes of DNN training, one should consider making static hyperparameters dynamic, thus allowing them to adapt to changes based on the state of training and deployment conditions.

Historically, one hyperparameter that has seen heavy optimization is the learning rate [64, 86, 87, 99, 107, 112, 113, 116, 122]. This is because of the high impact of learning rate on the model's final accuracy and its training speed. Adaptive learning rate schedulers such as Adam [64], AMSGrad [99], and AdamW [87] are now commonplace in DNN training and show the effectiveness of adapting to the current state of training. We apply adaptive hyperparameter tuning more broadly, extending this principle to other hyperparameters besides the learning rate, that have traditionally been considered static. In our work, we aim to provide evidence to support the following thesis statement:

Thesis: *Using runtime information to dynamically adapt hyperparameters that are traditionally static, such as the emphasis on individual training examples and the weights updated during transfer learning, can increase the efficiency of machine learning training and inference*

1.3 Case studies

In support of this thesis statement, we describe our experience with three case studies:

1.3.1 Training time adaptation of challenging example identification

First, we propose and analyze Selective-Backprop, a technique that accelerates deep learning training by prioritizing examples with high loss in an online fashion. Traditional machine learning methods consider all examples equally useful. As a result, algorithms like stochastic gradient descent (SGD) train on each example in the dataset, every epoch. In Selective-Backprop, we relax this constraint by focusing our training on examples that are relatively more useful. We hypothesize that challenging examples should be prioritized as they have more to teach the network (i.e. they lead to a larger gradient to apply). However, it is difficult to know in advance which examples will be challenging for our network to classify. Further the set of challenging examples will fluctuate over the course of training.

In Selective-Backprop, we consider the examples considered challenging as a tunable hyperparameter. We use the loss calculated from a training example as a signal of its usefulness during training. Selective-Backprop then decides for each example whether to use it to compute gradients and update parameters, or to skip immediately to the next example. Consequently, Selective-Backprop prioritizes examples that have more to teach the network. This choice is adaptive because which examples are considered to be "useful" will vary over time. Evaluation on CIFAR10, CIFAR100, and SVHN, across a variety of modern image models, shows that by reducing the number of computationally-expensive backpropogation steps, Selective-Backprop converges to target error rates up to 3.5x faster than with standard SGD and between 1.02–1.8x faster than a state-of-the-art importance sampling approach. We also demonstrate further acceleration of 26% by using stale losses to determine example selection.

1.3.2 Prioritization of challenging examples during training

Next, we explore AdaptSB, a variant of Selective-Backprop that dynamically adapts how we prioritize challenging examples. In Selective-Backprop, the prioritization for different degrees of example difficulty is fixed (i.e., an example with a loss in the 50th percentile of historical losses is always chosen 50% of the time). However, the optimal prioritization depends on the dataset and the training time. For example, Selective-Backprop assumes that the more challenging an example is, the more it should be prioritized. This holds true for well-curated datasets, where even high-loss examples are informative and representative. However, datasets with noise or mislabeling contain misleading examples that should be de-emphasized during training.

In AdaptSB, we treat the priority assigned to different classes of examples as a tunable hyperparameter. We model the prioritization assignment as a function between how challenging the example is, specifically the percentile of historical losses an example's loss represents, and its likelihood for selection. We find that by adapting the priority function to the training dataset, AdaptSB outperforms Selective-Backprop on datasets with label error. We also show that the priority function should be adapted over time. The priority function that leads to the most sample efficiency changes over the course of training, as suggested by techniques proposed in the field of curriculum learning [9, 70].

1.3.3 Inference time adaptation of the degree of DNN specialization

Finally, we explore Mainstream, a new video analysis system that jointly adapts concurrent applications sharing fixed edge resources to maximize aggregate result quality. In Mainstream, we introduce a technique for identifying and eliminating redundant computation between concurrently deployed applications. Each application relies partly on a DNN shared between the other applications as well as a specialized, task-specific, DNN. We show that the optimal allocation of resources and processing rate of each application depends on deployment conditions (e.g., compute capabilities of the edge device and the currently deployed applications).

Based on the available resources and mix of applications running on an edge node, Mainstream automatically determines at deployment time the right trade-off between using more specialized DNNs to improve per-frame accuracy, and keeping more of an unspecialized base model (shared between all deployed applications) to reduce resource contention and process more frames per second. Experiments with several datasets and event detection tasks on an edge node confirm that Mainstream greatly improves mean event detection F1-scores relative to a static approach of retraining only the last DNN layer and sharing all others ("Max-Sharing") by 71%, or to the common approach of using fully independent per-application DNNs ("No-Sharing") by 29X.

1.4 Thesis contributions and outline

We make three key contributions in this thesis.

1. Our first contribution improves training efficiency by reducing the time spent in the computationally expensive backwards pass. Big data and deep learning have brought the emergence of ever-larger labeled training datasets. While larger datasets enable models that to better generalize and that are robust to overfitting, it is no longer feasible to manually sanitize production datasets or to train through entire datasets in a brute force fashion. Instead, many design decisions are arising to determine which examples to include in the dataset [29, 29, 31, 106, 106, 108, 117], which examples to prioritize [6, 9, 13, 32, 54, 55, 57, 60, 70, 78, 85, 89, 101, 105, 109, 126], and which examples to augment [22, 37, 44, 69, 72]. We present the design and evaluation of Selective-Backprop and its variants StaleSB and AdaptSB, techniques for practical and lightweight importance sampling that automate many of the aforementioned design decisions. We include measurements showing that, compared to traditional training, Selective-Backprop and StaleSB reduce the time required to achieve target errors on CIFAR10, CIFAR100, and SVHN by up to 3.5x and 5x, respectively and 1.02–1.8x and 1.3–2.3x compared to a state-of-the-art importance sampling approach introduced by [61]. We also show that AdaptSB is more robust to label error than Selective-Backprop and StaleSB

2. Our second contribution improves application performance of real-time video analysis application . Real-time video analytics poses unique deployment constraints. Inference is typically run on high-frame rate videos and often use resource constrained edge devices to reduce end-to-end latency. Furthermore, these edge devices can be shared across tenants and applications. Designing DNNs for real-time video analysis requires careful optimization to the deployment conditions. We present the design and evaluation of Mainstream, a video analytics framework for automatically deploying the right degree of specialization for a DNN. This dissertation highlights the critical importance of reducing aggregate per-frame CPU work of multiple independently developed video processing applications via stem-sharing and identifies the goodness trade-off between per-frame quality and the frame sampling rate dictated by the degree of DNN specialization (and thus the amount of sharing).

3. Our third contribution is the analysis of three case studies on the importance of dynamically optimizing hyperparameters to changing conditions in the context of deep learning training and inference. These case studies use Selective-Backprop to show the efficacy of dynamically determining what examples from the training dataset are challenging to the network; AdaptSB to show what to do with those challenging examples; and Mainstream to show how much of the DNN to specialize during training. In all three cases we see a significant benefit to dynamically optimizing the hyperparameters to changes in training and inference conditions.

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the

current state of deep learning, an overview of deep learning training, and describes the application of image classification to edge video analytics. Chapter 3 describes Selective-Backprop and Chapter 4 describes its variant AdaptSB. In Chapter 5, we switch gears to optimizing for DNN inference, and describe Mainstream. Finally, Chapter 6 concludes the dissertation with a discussion on lessons learned and possible future research directions.

Chapter 2

Background

Deep learning refers to a family of machine learning methods based on artificial neural networks, a class of machine models vaguely inspired by biological neural networks. A machine learning *model* is a parameterized function that performs a task. *Training* is the process of learning parameter values (called weights) such that the model will approximate the desired function with some measure of accuracy (Section 2.2.1). For example, when training an image classifier, one might examine labeled input images and use gradient descent to find a set of weights that minimizes a loss function over the labels. Using the trained model to find the function's output given a new, unlabeled input is called *inference* (Section 2.2.4).

DNNs refer to a type of neural network architecture, used for problems which require a large input space, such as the pixels of an image. A DNN can be represented by a graph where nodes are organized into *layers*; each node computes a function of its inputs, which are outputs from the previous layer. The "deep" in DNNs refers to their many *hidden layers*. The example DNN in Figure 2.3 represents an DNN that performs image classification (Section 2.1.1). It takes an image as input, consists of three hidden layers, and outputs the likelihood the example belongs to each of four different classes. Increasingly, successful applications of DNNs have largely been the result of building models with more layers that take larger vectors of inputs [42, 68, 111, 114].

In this section, we describe the relevant background for our work. We start with an overview of deep learning training (Section 2.2.1) and inference (Section 2.2.4), including a description of modern approaches to DNN training and where the computational bottlenecks lie (Section 2.2.1). We also introduce our motivating applications of image classification, video analysis, and event detection (Section 2.1) and discuss the trade-offs of performing these applications in centralized datacenters or on edge devices near the data source.

Architecture	Number of Layers	ImageNet Top-1 Accuracy (%)
InceptionV3	314	78.0
MobileNets-224	84	70.7
ResNet-50	177	75.6

Table 2.1: Top-1 accuracy of three neural networks architectures trained on the ImageNet dataset.

2.1 Motivating applications

DNNs are used in a variety of tasks such as human action recognition [110], object detection [34], scene geometry estimation [30], face recognition [115] and neural machine translation [121]. While we believe that the technique of dynamically tuning hyperparameters should be applied broadly to deep learning, in our case studies we focus on the applications of image classification and streaming event detection on edge devices.

2.1.1 Image classification

Image classification aims to assign one label from a set of categories or *classes* to each image. For example, a 4-class classifier takes an input image and returns a 4-item vector of probabilities representing the likelihood that the image belongs to each class (Figure 2.3). *Top-N accuracy* is the probability that the correct label is among the top-N highest probability output labels. So, Top-1 accuracy indicates the fraction of images that the model classifies correctly. We refer to this metric as the *per-frame accuracy* in the context of video classification. Popular neural network architectures for image classification include ResNet [42], InceptionV3 [114], and MobileNets [46]. Table 2.1 describes these three neural networks and their Top-1 accuracy achieved on the ImageNet dataset [23]. Networks trained on ImageNet are popular base-DNNs for image classification tasks.

2.1.2 Streaming event detection

The increasing rate of video acquisition, combined with rapid progress in the field of computer vision, makes video analytics a fundamental building block of emerging applications such as smart cities, autonomous vehicles, and search-and-rescue drones. The volume of video data generated prohibits manual discovery as a scalable solution. Instead, computer vision powered by deep learning automates the process of extracting value and insights from video.

This dissertation focuses on applications that use image-classification DNNs to perform event detection. We define an event as a contiguous group of frames containing some visible phenomenon that we are trying to identify: e.g., a cyclist passing by, or a puff of smoke being emitted.

Figure 2.1: Example computation pipeline for event detection.

One way of doing event detection is to perform image classification across a sequence of frames. An event is detected if at least one of the contiguous frames is sampled, analyzed, and correctly labeled. Previous works [73] have also used this existence metric to measure recall and precision of range-based queries. (Event detection is not to be confused with object detection, where the goal is to locate an object in a single frame. Indeed, object detection is another way of performing event detection.) We evaluate event classification applications by measuring the *event F1-score*, the harmonic mean between *event recall* and *event precision*. The event recall reports the proportion of ground truth events identified. The event precision reports the proportion of classified events that are true positives. Note that these metrics are relative to the detection of events across multiple frames and are distinct from per-frame metrics (e.g., Top-1 accuracy).

Figure 2.1 shows a typical computation pipeline for an image classification application. The sensor or camera generates frames and sends them to the edge device for processing. The frame is decoded and then preprocessed with image transformation such as cropping and color normalization. The processed frame is then inputted into the DNN for classification. Although frame ingest and image preprocessing are necessary stages of computation, they are low cost and easily shared between concurrent applications. DNNs, on the other hand, are typically unique to each application and computationally expensive: in one image classification application we run, the DNN inference incurs 25X more latency than the preprocessing steps.

2.1.3 Video processing at the edge

Video analytics bring about two systems challenges. First, analyzing a high-frame rate video in real-time requires high throughput from our inference engine. Second, many applications require low response times (e.g. for real-time actuation). These goals are in tension given the nature of video camera deployments (Figure 2.2). In a typical deployment, cameras and sensors capture videos on the edge and have access to closeby compute with limited computational resources. The edge nodes can send data to a datacenter, which has heavy computational resources but is far away.

Back-hauling video from the edge to the datacenter gains advantages from resource availability, centralization, and economies of scale that come with cloud computing. However, the bandwidth required and the latency cost is often prohibitive. For example, HD cameras for intersections and other key points in a medium-sized city may generate tens of gigabytes of highly-compressed video every second. Only infrastructure-rich deployments have the capacity to ship this video back to the datacenter. A deployment of HD video cameras at every inter-

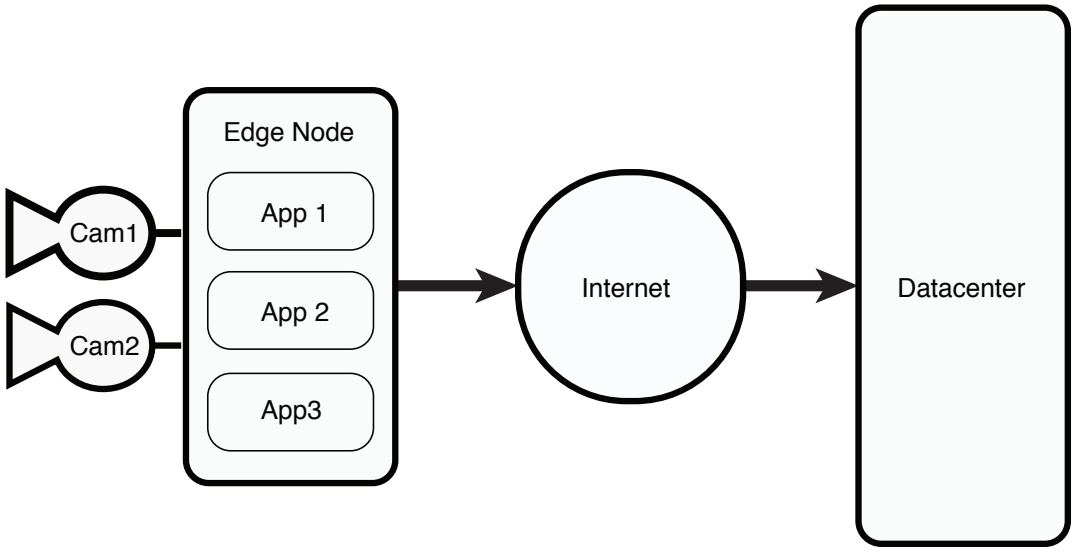


Figure 2.2: Example video analytics edge deployment.

section, police vehicle, and public bus in a medium-sized city may generate tens of gigabytes of highly-compressed video every second. Furthermore, if the analysis cannot tolerate lossy video compression artifacts [27], the bandwidth requirements swell to terabytes per second. Though it may be technically possible to provide such ingress data rates to a datacenter, it is impractical to deploy and dedicate a network with such high bandwidth from widely distributed sources to the data center. Mobile nodes (e.g., drones and cars) that rely on wireless communication have even less bandwidth at their disposal. Shipping data to the datacenter can also be prohibitively slow. Applications like autonomous driving require sub-millisecond latency for actuation and cannot tolerate a roundtrip between the edge and the cloud.

Moving compute to the data is a classic alternative to moving large amounts of data. Computing at the edge can also reduce cost: placing additional processing at the cameras may be cheap relative to the total deployment cost, including acquiring rights-of-way, truck-rolls/installation labor, and high-quality weather-proof cameras. A key challenge remaining is to run computation-intensive DNNs at a high throughput on resource-constrained edge devices.

To achieve the efficiency and scaling on edge deployments of DNNs, one must take into account form factor, cost, and energy consumption of the network as well as the hardware resources available. A variety of software optimizations can make inference more efficient, including the design of lightweight networks [46, 53], post-processing networks via weight quantization, weight sharing, and network pruning [38, 39, 63]. On the hardware side, edge devices for deep learning are emerging such as the Coral Accelerator from Google and Intel's Neural Compute Stick 2. At the time of writing, Google Coral Edge TPU is capable of performing 4 trillion operations per second (TOPS), using 0.5 watts for each TOPS [82]. While these approaches reduce the end-to-end latency and power consumption of inference, another complementary goal is to provide scalabil-

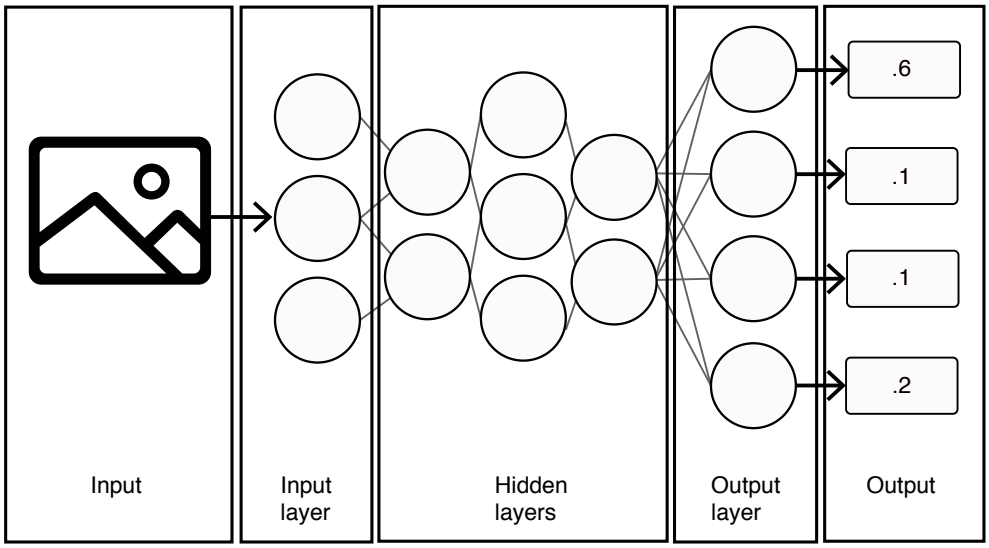


Figure 2.3: Example image classification neural network.

ity in edge deployments. Scalability of DNN deployments is increasingly important as a modern edge setup are expected to support more and more applications at a given time (Figure 2.2). For instance, a smart camera deployed in an intersection of a smart city may be used simultaneously for traffic analysis, infrastructure monitoring, and pollution monitoring. New applications which take advantage of the generated camera stream may continue to arise. As more applications are deployed, resources become further contended and application performance suffers. We thereby target reducing the *marginal cost* of introducing additional applications to the edge device.

2.2 Selected performance aspects of deep learning

2.2.1 DNN training

Deep learning training requires solving a large, nonconvex optimization problem. The goal is to learn the millions of weights that parameterize a deep model. The optimization problem is formulated as a loss minimization problem, often minimizing the cross-entropy loss summed across a set of training examples, between the model's predictions and the ground truth labels. Most commonly, mini-batch stochastic gradient descent is used to perform the optimization. The training process iterates over a labeled dataset, with items being passed forward through the network to determine losses; gradients derived from these losses are then backpropagated to adjust DNN model weights. The process of deriving and applying the gradients is termed the *backwards pass* or *backpropagation*. Traditionally, all examples in the dataset are used every epoch, and training progresses over many epochs. After several epochs, the model is able to produce accurate outputs

Network	Batch size	Forward conv (ms)	Backward convs (ms)	% training spent in conv kernels
Resnet18	32	31.7	53.7	76.3
Resnet18	64	53.3	99.5	86.3
Resnet18	128	99.6	193.2	92.8
Resnet18	256	191.5	382.5	96.5
Resnet18	512	371.7	990.6	98.3
Squeezenet	32	33.8	56.2	81.4
Squeezenet	64	62.2	105.8	88.6
Squeezenet	128	120.1	209.8	93.5
Densenet121	32	115.0	193.8	74.5
Resnet101	32	156.3	277.4	84.1
Resnet101	64	278.4	535.3	90.6

Table 2.2: Time spent in executing convolutions during ImageNet training running on a NVIDIA TitanX. Setup using PyTorch 0.4.0 and NVIDIA cuDNN 7.

for many items in the training set.

2.2.2 Backwards pass bottleneck

For general-purpose hardware and training frameworks, the backwards pass is typically the computational bottleneck. A standard image classification network is primarily composed of convolutional layers. Each layer performs a *convolution* operation, which essentially calculates dot products between a set of weights known as *filters* or *kernels*, and local regions of the input. The operation can be implemented as a single large matrix multiply, and thus can be parallelized using a GPU. We profile the time spent performing three convolutions (one for the forward pass and two for the backwards pass) on a modern deep learning setup. We train various modern DNN models using ImageNet on a NVIDIA TitanX. For large batch sizes, performing convolutions on the GPU can account for 98% of the time in training ImageNet (Table 2.2). The backwards pass incurs approximately twice the forward pass cost. This is because during the forward pass, we calculate one convolution per convolutional layer, whereas in the backwardw pass we perform two: one convolution to calculate the gradients with respect to the input data and another with respect to the layer weights [8]. Forward passes can be performed either for training or inference, the former requiring storing data to perform the subsequent backwards pass. The latency of the two types of inference are comparable (Table 2.3).

Network	Batch size	Forward training conv (ms)	Forward inference convs (ms)
Resnet18	32	31.7	30.1
Resnet18	64	53.3	52.2
Resnet18	128	99.6	98.6
Resnet18	256	191.5	189.4
Resnet18	512	371.7	368.5

Table 2.3: Comparison between the latency of an training vs. inference forward pass where data is not kept for a subsequent gradient calculation.

2.2.3 Transfer learning vs. training from scratch

Training deep models with millions of parameters is notoriously hard. The success of these models has hinged critically upon the arrival of very large, labeled datasets for training [3, 23, 77]. However, one often lacks sufficient labeled data or computational resources to train such a model. Therefore, most developers of applications using DNNs do not design their own networks. Rather they reuse published networks that have been shown effective in the general domain, and retrain them for the specific task at hand. In fact, the general practice is not to fully train weights from scratch (i.e., starting with randomized weights), but to adapt a network model that was previously trained on a similar task. This approach, called transfer learning, can reduce the amount of labeled data needed for training as well as total training time by several orders of magnitude relative to fully training the network from scratch. As an example, transfer learning helps achieve 73% accuracy on human action recognition as opposed to 53% accuracy when training from scratch [110].

The intuition behind transfer learning is that a DNN which has, for example, been trained to classify cars in images, actually devotes most of the earlier stages to transforming the raw pixel inputs into some hidden internal feature representations that are robust to perspective, lighting, distortions, and occlusions. These are combined in various ways to finally perform the task-specific car detection in the final layers. Much of this machinery, with little change, may also be very useful in detecting other objects, such as buses, requiring mostly modified parameters in the final layers to perform the new task. Thus, for example, transfer learning a bus detector from a car detector can use far fewer training examples and training time than training from scratch.

2.2.4 DNN inference

After training, the DNN model is deployed for inference, whereby data is inputted into the trained model to infer a result. While inference is not as computationally expensive as training, it still requires more compute relative to traditional machine learning models. Furthermore, training is done in the datacenter with access to many GPUs or even TPUs, whereas inference is often

performed on an edge device with only a low power GPU, CPU, or embedded device. The challenge with inference then is to meet often strict latency and throughput service level agreements on a variety of hardware and deployment setups. Inference using InceptionV3, a popular image classification network, runs close to 10x faster on a NVIDIA 1080ti compared to an Intel i7-8700k CPU and 30x faster than using the Intel Neural Compute Stick [76].

Chapter 3

Adaptive Importance Sampling for Training Large Datasets

Next, we explore Selective-Backprop, a technique that accelerates the training of deep neural networks (DNNs) by prioritizing examples with high loss at each iteration. Selective-Backprop uses the output of a training example's forward pass to decide whether to use that example to compute gradients and update parameters, or to skip immediately to the next example. By reducing the number of computationally-expensive backpropagation steps performed, Selective-Backprop accelerates training. Evaluation on CIFAR10, CIFAR100, and SVHN, across a variety of modern image models, shows that Selective-Backprop converges to target error rates up to 3.5x faster than with standard SGD and between 1.02–1.8x faster than a state-of-the-art importance sampling approach. Further acceleration of 26% can be achieved by using stale forward pass results for selection, thus also skipping forward passes of low priority examples. The implementation of Selective-Backprop is open-source.

3.1 Overview

While training neural networks (e.g., for classification), computational effort is typically apportioned equally among training examples, regardless of whether the examples are already scored with low loss or if they are mis-predicted by the current state of the network [45]. In practice, however, not all examples are equally useful. As training progresses, the network begins to classify some examples accurately, especially redundant examples that are well-represented in the dataset. Training using such samples may provide little to no benefit; hence, limited computational resources may be better spent training on examples that the network has not yet learned to predict correctly.

Figure 3.2 illustrates the redundancy difference between "easy" examples and "hard" examples. Figure 3.2a shows examples from CIFAR10 that consistently produce low losses over the course of training. Compared with examples that generate high losses (Figure 3.2b), the classes

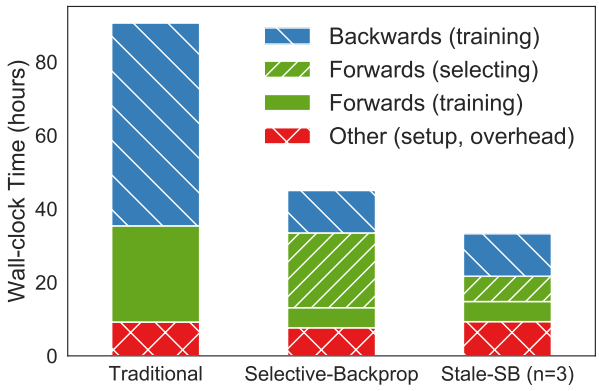
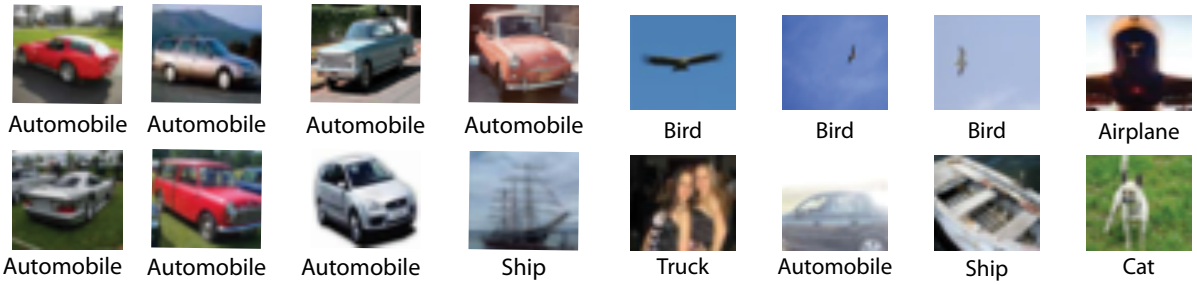


Figure 3.1: Comparison and breakdown of training time by Traditional training and proposed Selective-Backprop approaches, for training Wide-Resnet on SHVN until 1.72% error rate is achieved (1.2 times the final error of Traditional). SB accelerates training by reducing the number of computationally expensive backward passes. StaleSB further accelerates training by sometimes reusing losses calculated in the prior epoch for example selection.

of low-loss examples are easily distinguishable. Qualitatively, we find that low-loss examples consist of objects that are taken at common camera angles (e.g. taken from the front-right corner of a car, or the broadside of a ship). High-loss examples are challenging for even humans to classify. In the fifth image of Figure 2b., the image of the truck is obstructed by two people. There are also many birds, ships, and planes represented in high-loss examples as they are hard to distinguish from each other.

Motivated by the hinge loss [102], which provides zero loss whenever an example is correctly predicted by sufficient margin, this chapter introduces *Selective-Backprop* (SB), a simple and effective sampling technique for prioritizing high-loss training examples throughout training. We suspect, and confirm experimentally, that examples with low loss correspond to gradients with small norm and thus contribute little to the gradient update. Thus, Selective-Backprop uses the loss calculated during the forward pass as a computationally cheap proxy for the gradient norm, enabling us to decide whether to apply an update without having to actually compute the gradient. Selective-Backprop prioritizes gradient updates for examples for which a forward pass reveals high loss, probabilistically skipping the backward pass for examples exhibiting low loss.

By reducing computation spent on low-loss examples, Selective-Backprop reaches a given target accuracy significantly faster. Figure 3.1 shows this effect for one experiment in our evaluation. As seen in the first stacked bar ("Traditional"), in which every example is fully trained on in each epoch, backpropagation generally consumes approximately twice the time of forward propagation [16]. In this experiment, Selective-Backprop (second stacked bar) reduces the number of backpropagations by ≈70% and thereby cuts the overall training time in half. Across a range of models and datasets, our measurements show that Selective-Backprop speeds training to target errors by up to 3.5x.



(a) Examples chosen least frequently by SB **(b)** Examples chosen most frequently by SB

Figure 3.2: Example images from CIFAR10

Given Selective-Backprop's reduction of backpropagations, over half of the remaining training time is spent on forward passes, most of which correspond to non-selected examples. These forward passes do play an important role, though, because the loss order of examples varies throughout training and varies more with Selective-Backprop. A model might generate relatively low loss on a given example after some training, but progressively higher losses on the same example if it is ignored for several epochs [45]. Selective-Backprop evaluates sampling probabilities on the basis of an up-to-date forward pass, ensuring its assessment of the network's performance on the example is out of date.

The number of forward passes can be reduced, however, by allowing for some staleness in the selection process. One simple approach, for which we call the corresponding SB variant StaleSB (third stacked bar), is to perform forward passes to inform selection only every n^{th} epoch. In intervening epochs, StaleSB uses the results of the most recent previous forward pass of an example for selection, though it needs an up-to-date forward pass for the training of selected examples. For n=3, Figure 3.1 shows that StaleSB avoids approximately half of all forward passes, reducing training time by 26% relative to Selective-Backprop with minimal loss in final accuracy. Although our experiments show StaleSB captures most of the potential reduction, we also discuss other approaches to reducing selection-associated forward pass time.

Selective-Backprop requires minimal modifications to existing training protocols, applies broadly to DNN training, and works in tandem with data augmentation, cutout, dropout, and batch normalization. Our experiments show that, without changing initial hyperparameters, Selective-Backprop and StaleSB can decrease training times needed to achieve target error rates. Across a wide range of configuration options, including training time budgets, Selective-Backprop and StaleSB provide most of the Pareto-optimal choices. Sensitivity analyses also show that Selective-Backprop is robust to label error and effective across a range of selectivity settings.

This section makes three primary contributions: (1) The design and evaluation of Selective-Backprop and StaleSB, practical and effective sampling techniques for deep learning; (2) Measurements showing that, compared to traditional training, SB and StaleSB reduce the time required to achieve target errors on CIFAR10, CIFAR100, and SVHN by up to 3.5x and 5x, respectively; and (3) Comparison to a state-of-the-art importance sampling approach introduced in [61], showing

that SB and StaleSB reduce training times needed to achieve target accuracy by 1.02–1.8x and 1.3–2.3x, respectively.

3.2 Related work

Several papers propose to reduce variance and accelerate neural network training. The key idea of these techniques is to bias the selection of examples from the training set, selecting some examples with higher probability than others. A common approach is to use importance sampling, where the goal is to more frequently sample rare examples that might correspond to large updates. Classic importance sampling techniques weight the items inversely proportional to the probability that they are selected, producing an unbiased estimator of the stochastic gradient. Previous approaches use importance sampling to reduce the number of training steps to reach a target error rate in classification [32, 57, 85], reinforcement learning [105], and object detection [78, 109]. These works generate a distribution over a set of training examples and then train a model by sampling from that distribution with replacement. They maintain historical losses for each example, requiring at least one full pass on all data to construct a distribution, which they subsequently update over the course of multiple training epochs. Consequently, these approaches must maintain additional state proportional to the training set in size and rely on hyperparameters to modulate the effects of stale history. In contrast, the base Selective-Backprop approach does not require such state, though some optimization options do.

The approach most related to ours [61] also removes the requirement to maintain history, providing a fully-online approach to importance sampling for classification. Similar to Selective-Backprop, it uses extra forward passes instead of relying on historical data, allowing it to scale more easily to large datasets, and it also makes decisions based on an up-to-date state of the network. Their sampling approach, however, predetermines the number of examples selected per batch, so example selection is dictated by the distribution of losses in a batch. It also relies on a variable starting condition. We compare against this technique in Section 3.6.

Importance sampling and curriculum learning are common techniques for generalizing DNNs. Differing philosophies motivate these approaches: supplying easy or canonical examples early in training as in self-paced learning [9, 70], emphasizing rare or difficult examples to accelerate learning, or avoiding overfitting [6, 54, 60], targeting marginal examples that the network oscillates between classifying correctly and incorrectly [13], or taking a black-box, data-driven, approach [55, 89, 101]. These works improve target accuracy on image classification tasks, and datasets with high label error [55, 101]. These techniques, however, do not target and analyze training speedup [9, 13, 60, 70], often adding overhead to the training process by, e.g., training an additional DNN [55, 60, 126] or performing extra training passes on a separate validation set [101]. For instance, [126] requires running an additional DNN asynchronously on separate hardware to speed up training.

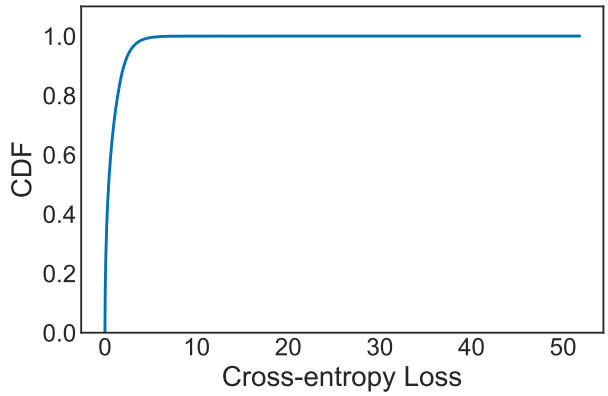


Figure 3.3: Snapshot of a CDF of cross-entropy losses whentraining MobilenetV2 with CIFAR10

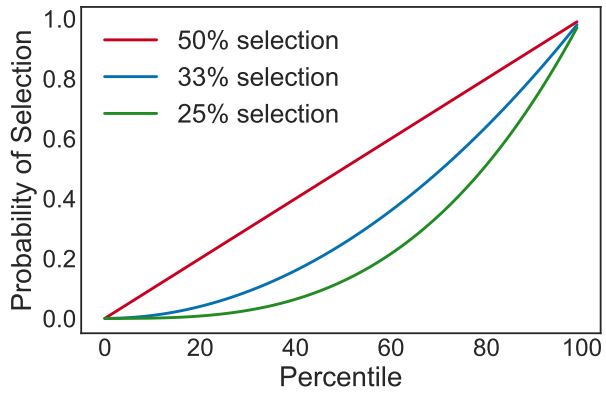


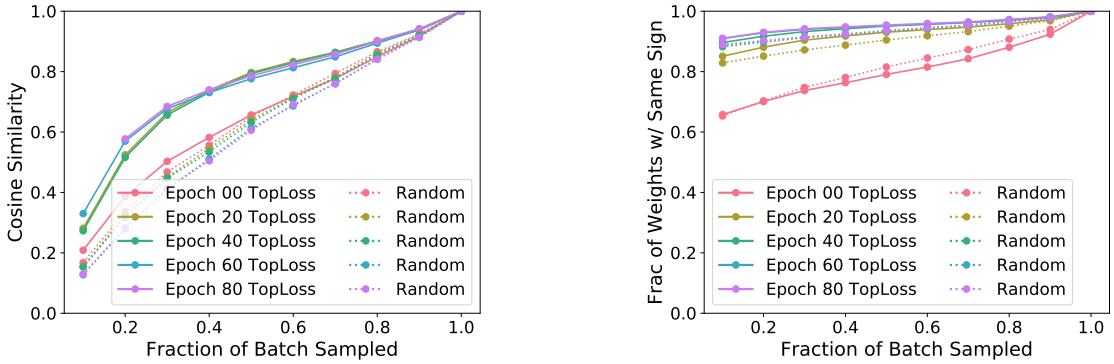
Figure 3.4: Selective-Backprop calculates the probability of selection using $\mathcal{L}(d)$ as a percentile of the R most recent losses

3.3 Loss-based sampling with Selective-Backprop

3.3.1 Background

Selective-Backprop can be applied to standard mini-batch stochastic gradient descent (SGD), and is compatible with variants such as AdaGrad, RMSprop, and Adam that differ only in learning rate scheduling. The goal of SGD is to find parameters $\mathbf{w}^*$ that minimize the sum of the losses $\mathcal{L}$ for a model $f(\mathbf{w})$ with d parameters over all points (indexed by i) in a dataset $\mathcal{D}$ consisting of n examples $(\mathbf{x}_i, y_i)$.

$$\mathbf{w}^* = \text{argmin}_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$$



(a) Cosine similarity

(b) Fraction of gradient components with same sign

Figure 3.5: Similarity between gradients calculated on the original batch and subsampled batches when training MobilenetV2 on CIFAR10. After the first epoch, high-loss examples are more similar than random subsampling by both cosine similarity and fraction of weights with same sign.

SGD proceeds in a number of iterations, at each step selecting a single example i and updating the weights by subtracting the gradient of the loss multiplied by a step-size parameter η.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \mathcal{L}(f_{\mathbf{w}_t}(x_i), y_i)$$

In minibatch gradient descent, at each step, one selects a subset of examples $\mathcal{M}_t$, often by sampling from $\mathcal{D}$ at random without replacement, traversing the full training set once per epoch, applying the update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \sum_{x_i, y_i \in \mathcal{M}_t} \nabla_{\mathbf{w}} \mathcal{L}(f_{\mathbf{w}_t}(x_i), y_i)$$

We refer to this approach as Traditional.

3.3.2 Selective-Backprop

Selective-Backprop also traverses the training set once per epoch, but, like other selection-based acceleration techniques, it generates batches using a non-uniform selection criteria designed to require fewer backward pass calculations to reach a given loss. To construct batches, Selective-Backprop selects each example with a probability that is a function of its current loss as determined by a forward pass through the network: $\mathcal{P}(\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i))$.

In each epoch, Selective-Backprop randomly shuffles the training examples $\mathcal{D}$ and iterates over them in the standard fashion. However, for each example i, after computing a forward pass to obtain its loss $\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)$, Selective-Backprop then decides whether to include the example for a gradient update by selecting it with probability $\mathcal{P}(\mathcal{L})$ that is a function of the current loss. Selecting a sufficient number of examples for a full batch ($\mathcal{M}_t$) for a gradient update typically

Algorithm 1 Selective-Backprop training loop.

```
function TRAIN_EPOCH(data, bSize)
    batch_bp ← []
    for all batch_fp in data.getBatches(bSize) do
        losses ← net.Forward(batch_fp)
        for i ← 0 to bSize − 1 do
            example ← batch_fp_i
            prob ← sb.CalcProb(losses_i)                ▷ See Eqn. 3.1
            if choose(prob) then
                batch_bp.append(example, losses_i)
            end if
            if batch_bp.size == bSize then
                net.Backward(batch_bp)
                batch_bp ← []
            end if
        end for
    end for
end function
```

requires forward pass calculations on more than $\mathcal{M}_t$ examples. After collecting a full batch, SB updates the network using gradients calculated based on this batch. Alg. 1 details this algorithm.

Setting the selection probability to 1 for all examples expresses standard minibatch SGD. For Selective-Backprop, we develop an intuitive heuristic whereby examples with higher loss are more frequently included in updates (Figure 3.2b), while those with the lower losses (Figure 3.2a) are included less frequently. Our experiments show that suppressing gradient updates for low-loss examples has surprisingly little impact on the updates. For example, we empirically find that the sign of over 80% of gradient weights is maintained, even when subsampling only 10% of the data with the highest losses (Fig. 3.5b). Since recent research has demonstrated that the sign of the gradient alone is sufficient for efficient learning [10], this bodes well for our method. Moreover, gradients calculated with only the highest loss examples maintain higher cosine similarity to those calculated with all examples as compared to randomly subsampling examples in a batch (Fig. 3.5a).

Alg. 1 also details our heuristic for setting $\mathcal{P}(\mathcal{L})$. We set $\mathcal{P}(\mathcal{L})$ to be a monotonically increasing function of the CDF of losses across the example set. In Figure 3.3, we show an example of historical losses snapshotted during training. Because recomputing the complete CDF after each update is not practical, we approximate the current CDF using a running tally of the losses of the last R examples, denoted by CDF_R:

$$\mathcal{P}\Big(\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)\Big) = \Big[\text{CDF}_R\Big(\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)\Big)\Big]^{\beta}, \tag{3.1}$$

where $\beta > 0$ is a constant that determines Selective-Backprop's level of selectivity and thus

allows us to modulate the bias towards high-loss examples where larger values produce greater selectivity (Figure 3.4). We include a sensitivity analysis of β in Section 3.6.

3.4 Reducing selection overhead

3.4.1 StaleSB reuses previous losses

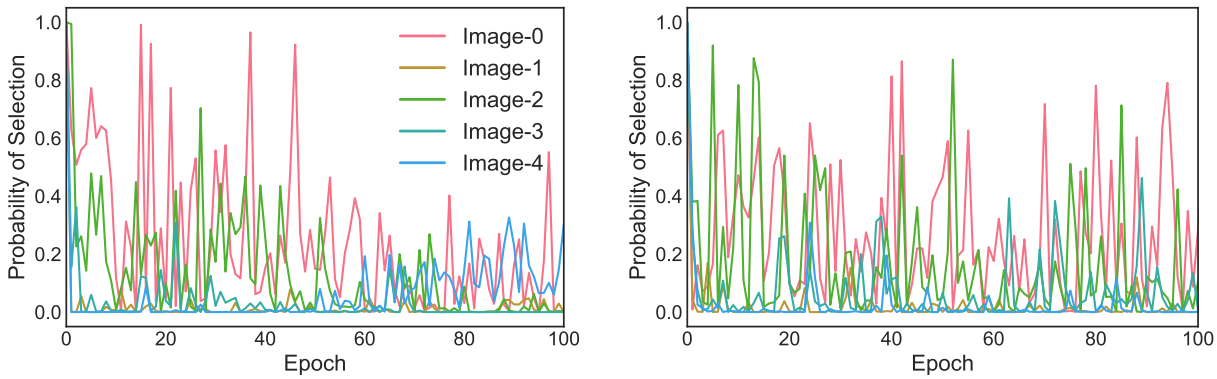
Selective-Backprop accelerates training by reducing the number of backward passes needed to reach given levels of loss. In our experiments (Section 3.6), we find that after reducing backward passes with Selective-Backprop, the largest remaining fraction of training time is the full (original) complement of forward passes used to select the Selective-Backprop batches. We distinguish these forward passes from the forward passes used for training by referring to them as "selection passes" in the rest of the chapter. This section describes four approaches to reducing the time spent in selection passes of Selective-Backprop training, thereby further reducing overall training time.

Re-using previous losses. Selective-Backprop's selection pass uses the latest model parameters to compute up-to-date losses for all training examples considered. We define and evaluate a Selective-Backprop variant called StaleSB that executes selection passes every n^{th} epoch. The subsequent $(n-1)$ epochs reuse the losses computed by the previous selection pass to create the backprop batch. The losses are reused in the following epoch(s), but only for batch formation. Intuitively, if an example is deemed important in a given selection pass, it will also have a high probability of being selected in the next $(n-1)$ epochs. StaleSB with $n = 1$ is Selective-Backprop. We evaluate StaleSB in Section 3.6 and find that, typically, it reduces selection pass cost significantly without impacting final accuracy.

3.4.2 Further optimizations

Using predicted losses. Rather than simply re-using the loss from an earlier epoch for selecting examples, one could construct a Selective-Backprop variant that predicts the losses of examples using historical loss values. To make the problem easier, instead of predicting the loss directly, one could predict whether the loss is high enough to cross the threshold for selection. We evaluated various prediction approaches, including tracking a exponentially weighted average of historical losses and using historical losses to train a Gaussian process predictor. None outperformed the simpler StaleSB approach.

Pipelining loss computation. Given multiple computation engines, one could construct a Selective-Backprop variant that selects examples for batch $N+1$ on a separate engine while training with batch N is ongoing. Such an approach would require using losses computed from stale versions of the model, but could mask nearly all training delays for selection and could do so without the "history size" concerns that would arise for the above approaches when training with



(a) Without SB sampling.

(b) With SB sampling.

Figure 3.6: Select probabilities of five examples when training MobilenetV2 on CIFAR10. Each line represents one image. Likelihood of selection fluctuates more when sampling is introduced.

giant or continuous datasets. Running a separate model for loss computation would, however, introduce a new overhead of occasionally syncing the selection model to reflect changes to the training model.[1] This introduces a new trade-off between frequency of syncing the selection model and the amount of staleness introduced to the selection process.

Although one could use equivalent GPUs for such pipelining, it is unclear that this would be better than data-parallel training. Rather, we think the natural application of the pipelining approach would be in combination with the inference accelerators discussed next—that is, a low-cost inference accelerator could be used to compute losses for example selection, and then a powerful compute engine could be used for training on batches of selected examples.

Inference accelerators. For general-purpose hardware and training frameworks, the cost of the backward pass is approximately twice the forward pass cost. This is because during the forward pass, we calculate one convolution per convolutional layer, whereas in the backward pass we perform two: one convolution to calculate the gradients w.r.t the input data and another w.r.t to the layer weights [8]. But, a variety of inference acceleration approaches, such as reduced precision or quantization, may enable specialized hardware accelerators to run forward passes ≈10x faster than a backward pass on a modern GPU [58]. Since SB selects examples by running a forward pass, it can use such accelerators. Although aggressive forward-pass acceleration can affect the outcome of training, use of inference acceleration for Selective-Backprop's selections may not have the same negative consequences. We leave exploration of this approach to reducing selection time to future work, but include it in this list for completeness.

[1]Using modern deep learning frameworks such as PyTorch, we found that copying a new model and moving it to a second device can take up to a minute.

3.5 Implementation

We built prototypes of SB for PyTorch 0.4.1 and Keras 2.2.4; the Section 3.6 evaluation is based on the former.

To add SB into existing training code, we introduce a mathematically simple probabilistic filtering step to training, which down-selects examples used for updates. Filtering starts by calculating the loss for each example using a forward pass. SB adds the loss to CDF_R (implemented using a bounded queue), and calculates what percentile of losses it represents. Using this percentile, SB calculates the selection probability $\mathcal{P}$, and probabilistically adds this example to the minibatch for training. We measure the overhead introduced by SB's selection step, excluding time spent in selection passes, to ≈3% of overall training time.

Selective-Backprop's lightweight filtering step is simple to implement and requires few changes to existing training code. In traditional setups, data is formed into minibatches for training. In SB, data is formed into selection minibatches and fed into SB's filtering mechanism. SB performs forward passes of selection minibatches ("selection passes"), forms a training minibatch of selected data examples, and passes it to the original training code. Therefore, the training code can be agnostic to SB's filtering mechanism, allowing SB to work in tandem with any training optimizer (e.g., SGD, Adam) and common optimizations such as batch normalization, data augmentation, dropout, and cutout.

Future implementation optimizations. Our SB implementation minimizes changes to existing code, and some obvious potential optimizations are not currently incorporated. For instance, in our implementation, two forward passes are performed for each selected example: one for selection and one for training. Unless selection passes are accelerated using reduced precision or quantization, which is not the case in our implementation, a more optimized SB implementation could cache the activations obtained from the selection passes to avoid doing extra forward passes for training, and thus eliminate the time spent in "Forwards (training)" for SB shown in Figure 3.1. Another optimization would use a minibatch size for selection that is larger than that of training, to reduce the number of selection passes needed to populate a training minibatch.

3.6 Evaluation

We evaluate Selective-Backprop's effect on training with modern image classification models using CIFAR10, CIFAR100, and SVHN. The results show that, compared to traditional training and a state-of-the-art importance sampling approach [61], SB reduces wall-clock time needed to reach a range of target error rates by up to 3.5x (Section 3.6.2). We show that by reducing the time spent in example selection, one can further accelerate training by on average 26% (Section 3.6.3). Additional analyses show the importance of individual SB characteristics, including selection of high-loss examples and robustness to label error (Section 3.6.4). Throughout the evaluation, we

also show that the speedup achieved by a SB depends the learning rate schedule, sampling selectivity, and target error. Section 3.6.5 shows that, across a sweep of configurations, the majority of Pareto-optimal trade-off points come from Selective-Backprop and StaleSB. We also provide a sensitivity analysis for SB in Section 3.6.4 and the results from two additional learning rate schedules in the appendix.

3.6.1 Experimental setup

We train Wide Resnet, ResNet18, DenseNet, and MobileNetV2 [43, 50, 104, 124] using SB, Traditional as described in Section 3.3, and our implementation of Kath18 [61] with variable starting, no bias reweighting, and using loss as the importance score criteria. We tune the selectivity of SB and Kath18 individually for each dataset. In our evaluation, we present the results of training Wide Resnet. To train Wide Resnet, we use the training setup specified by [26], which includes standard optimizations including cutout, batch normalization and data augmentation. We observe similar trends when using ResNet18, DenseNet, and MobileNetV2, and present those results in the appendix. In each case, we do not retune existing hyperparameters. We report results using the default batch size of 128 but confirm that the trends remain when using batch size 64.

CIFAR10. The CIFAR10/100 datasets [67] contain 50,000 training images and 10,000 test images, divided into 10 and 100 classes, respectively. Each example is a 32x32 pixel image. We use $batch\_size = 128$ and cutout of length 16. We use an SGD optimizer with $decay = 0.0005$. We train with two learning rate schedules. In the first schedule, we start with $lr = 0.1$ and decay by 5x at 60, 120, and 160 epochs. In the second schedule, we decay at 48, 96, and 128 epochs. We use 33% selectivity for SB, StaleSB, and Kath18. Training ends after 12 hours.

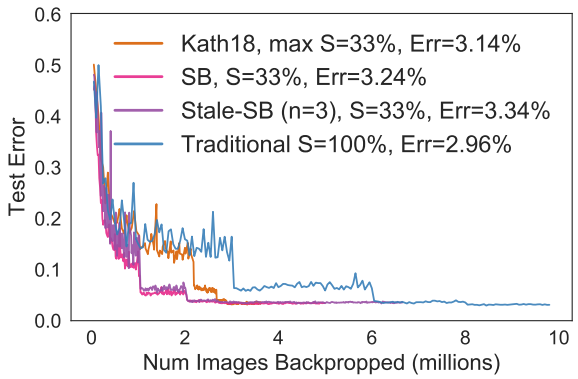
CIFAR100. We train on CIFAR100 using the setup specified by Devries [26]. We use $batch\_size = 128$. and cutout of length 8. We train with two learning rate schedules. First, we start with $lr = 0.1$ and decay by 10x at 60 and 120. In the second learning rate schedule, we decay at 48 and 96 epochs. We use 50% selectivity for SB, StaleSB, and Kath18. Training ends after 12 hours.

SVHN. SVHN has 604,388 training examples and 26,032 testing examples of digits taken from Street View images [92]. We initialize the learning rate to 0.1 and decay to 0.01 and 0.001 at epochs 5 and 10, respectively. We use $batch\_size = 128$ and cutout of length 20. We use 25% selectivity for SB and StaleSB, and 33% selectivity for Kath18. Training ends after 96 hours.

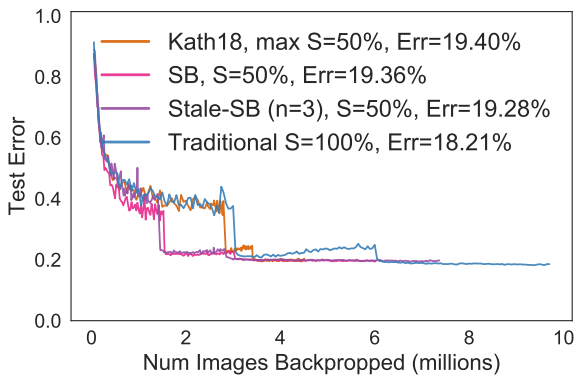
Hardware. We train CIFAR10 and CIFAR100 on servers equipped with 16-core Intel Xeon CPUs, 64 GB RAM and NVIDIA TitanX GPUs. We train SVHN on servers with four 16-core AMD Opteron CPUs, 128 GB RAM, and NVIDIA Tesla K20c GPUs.

3.6.2 Selective-Backprop speeds up training

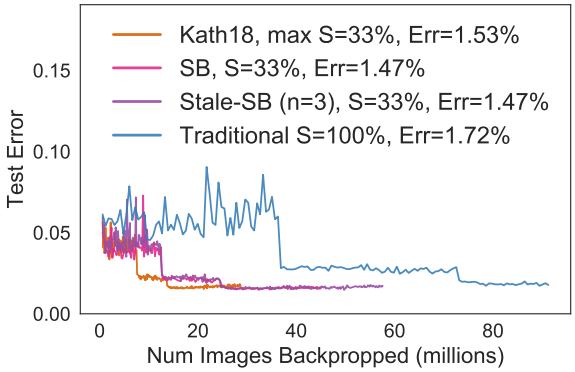
SB reduces training iterations to target error. SB probabilistically skips the backward passes of examples with low loss in order to learn more per example. Figure 3.7 shows that SB reaches



(a) CIFAR10



(b) CIFAR100



(c) SVHN

Figure 3.7: SB reduces training iterations to target error. S is the selectivity used, and Err is the final test error reached.

final or non-final target error rates with fewer training iterations (updates to the network). This can be seen by comparing the x-axis points at which lines for each approach reach a particular y-axis value. Note that we plot different y-axes for different datasets. Although the savings depends on the specific target test error rate chosen, one way to visualize the overall speedup across different target accuracy is by comparing the area under the three curves. SB reaches nearly every test error value with significantly fewer training iterations.

SB reduces wall-clock time to target error. Figure 3.8 shows error rate as a function of wall-clock time. SB speeds up time to target error rates by reducing backward passes, without optimizations to reduce selection time discussed in Section 3.6.3. Table 3.1 shows that for CIFAR10, SB reaches within 10%, 20%, and 40% of Traditional's final error rate 1.2–1.5x faster. For SVHN, SB provides a 3.4–5x speedup to reach 1.8%, 2.1%, and 2.4% error.

Intuitively, SB is most effective on datasets with many redundant examples or examples that are easy to learn. CIFAR100 is a more challenging dataset for sampling as there are fewer examples per class and therefore likely less redundancy. Despite this, SB reaches within 20% and 40% of the Traditional's final error rate 20% faster. However, it sacrifices a small amount of final accuracy for these speedups and does not reach within 10% of Traditional's final error rate in the allotted training time. Kath18 also accelerates training over Traditional by 0.8–3.4x. Similarly to SB, it is most effective on SVHN and least effective on CIFAR100, even leading to a small slowdown to certain target error rates. SB provides a speedup over Kath18 of 1.02–1.8x.

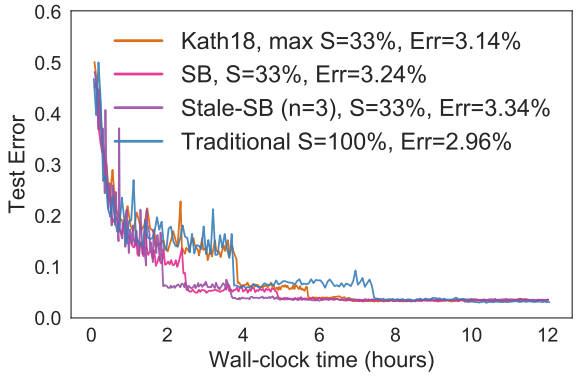
Selective-Backprop performs better on challenging examples. SB converges faster than Traditional by outperforming Traditional on challenging examples. Figure 3.9 shows an inverse CDF of the network's confidence in each ground truth label of the test set; the data represents a snapshot in time after training SB or Traditional for ten epochs. For each percentile, we plot the target confidence on the y-axis (e.g., the 20^{th} percentile of target confidences for SB is 55%). The network's classification is the class with the Top-1 confidence (using argmax). Therefore, we cannot infer the classification accuracy of an example solely from its target confidence (if the target confidence is $\leq$ 50%). In Figure 3.10, we also plot the accuracy of each percentile of examples. Generally, examples at lower percentiles are harder for the network to accurately classify. Using SB, the network has higher confidence and accuracy in these lower percentiles. For instance, among the examples at the 20th percentile of target confidences, 29% of these examples are classified correctly using SB while only 3% are classified correctly by Traditional. While this comes at the cost of confidence in higher percentile examples, test accuracy is not sacrificed. In fact, SB is able to generalize better across all examples of all difficulty levels.

3.6.3 Reducing selection times further speeds training

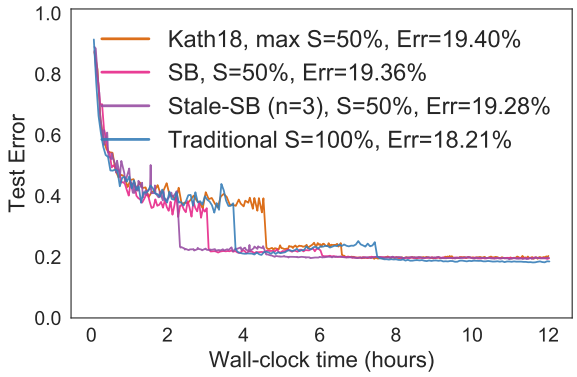
In Figure 3.11, we see that SB reduces total time to target error rates compared to Traditional by reducing the time spent in the backward pass. In Figure 3.11a and Figure 3.11b, both Traditional and SB take the same number of epochs to reach the target error rate. However, SB performs more overall forward passes. As described in Section 3.5, this is because we perform one selection

Dataset	Strategy	Final error of Traditional	Speedup to final error × 1.1	Speedup to final error × 1.2	Speedup to final error × 1.4
CIFAR10	SB	2.96%	1.4x	1.2x	1.5x
CIFAR10	StaleSB	2.96%	–	1.5x	2.0x
CIFAR10	Kath18	2.96%	1.4x	1.1x	1.3x
CIFAR100	SB	18.21%	1.2x	1.2x	1.2x
CIFAR100	StaleSB	18.21%	1.5x	1.0x	1.6x
CIFAR100	Kath18	18.21%	1.1x	0.8x	0.8x
SVHN	SB	1.72%	3.4x	3.4x	3.5x
SVHN	StaleSB	1.72%	4.3x	4.9x	5.0x
SVHN	Kath18	1.72%	1.9x	2.8x	3.4x

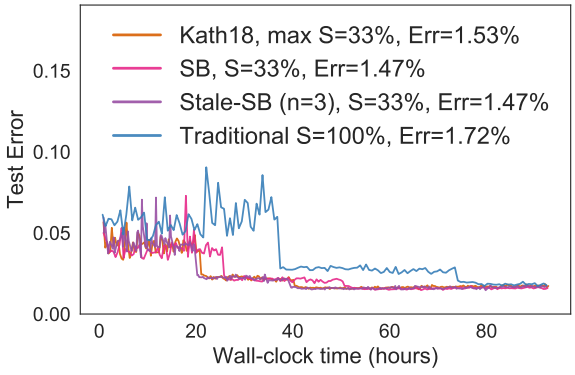
Table 3.1: Speedup achieved by SB and Kath18 over Traditional



(a) CIFAR10



(b) CIFAR100



(c) SVHN

Figure 3.8: SB reduces wall-clock time to target error. S is the selectivity used, and Err is the final test error reached.

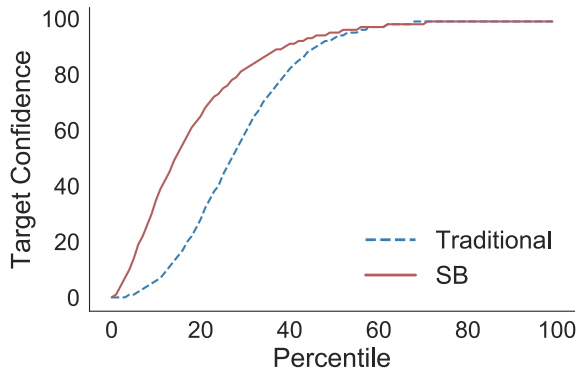


Figure 3.9: SB has higher confidence in harder examples with almost no cost of confidence in easy examples.

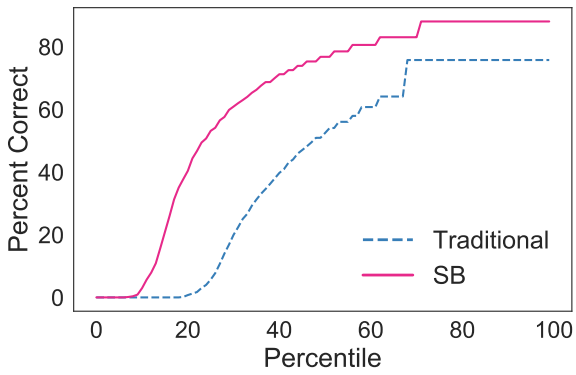
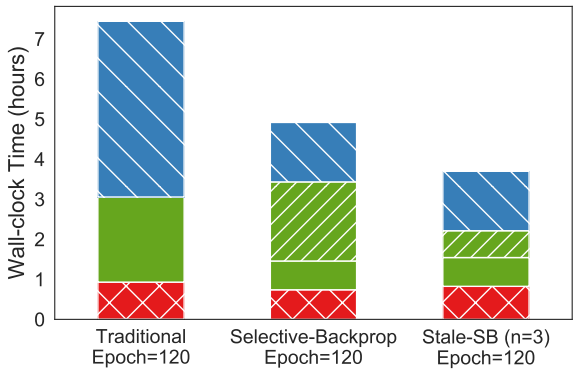
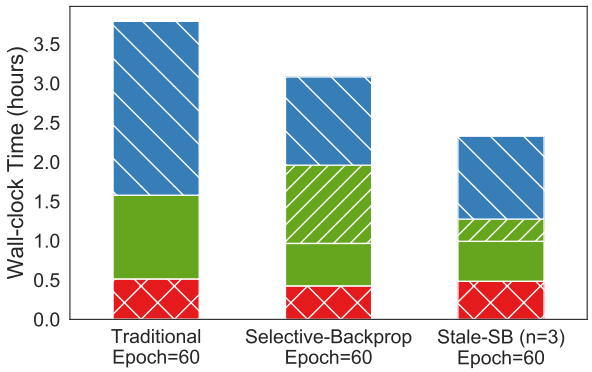


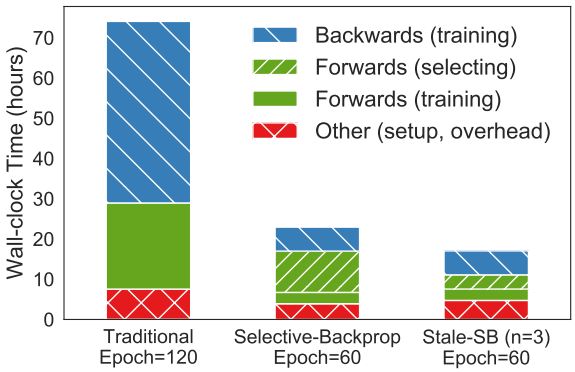
Figure 3.10: SB increases accuracy of harder examples, without sacrificing accuracy of easy examples.



(a) CIFAR10



(b) CIFAR100



(c) SVHN

Figure 3.11: SB reduces time spent in the backward pass in order to speed up time to the target error rate (in this case, 1.4x of Traditional's final error rate). StaleSB further accelerates training by reducing the time spent performing selection passes.

forward pass for each candidate example, plus one training forward pass for each selected example. The "Other" bar shown for each run includes per-run overheads (e.g., loading the dataset into memory and the network onto the GPU) and per-epoch overheads (e.g., evaluating test accuracy).

Using stale losses to reduce selection passes. After SB's reduction of backward passes performed, over half of the remaining training time is spent on forward passes. We evaluate StaleSB, which uses the losses of forward passes from previous epochs to perform selection. With StaleSB, we run fewer selection passes, running them only every $n = 2$ or $n = 3$ epochs. That is, if $n = 2$, an example that incurs a high loss has a high chance of being trained on in the next two epochs, instead of just one. In Figure 3.11, we see that StaleSB with $n = 3$ reduces the time spent performing selection passes by two-thirds, thereby further reducing the total wall-clock time. In Figure 3.12, we see that the reduced number of total forward passes in StaleSB has little effect on final error. StaleSB's ability to reduce selection passes while maintaining test accuracy leads to the end-to-end speedups shown in Table 3.1. On average, StaleSB with $n = 3$ reaches target error rates 26% faster than SB. With $n = 3$, we believe StaleSB captures most of the benefits of reducing selection passes, though we have not yet experimented with values of $n > 3$.

3.6.4 Selective-Backprop sensitivity analysis

SB is robust to modest amounts of label error. One potential downside of SB is that it could increase susceptibility to noisy labels. However, we show that on SVHN, a dataset known to include label error [95], SB still converges faster than Traditional to almost all target error rates. We also evaluate SB on CIFAR10 with manually corrupted labels. Following the UniformFlip approach in [101], we randomly flip 1% (500 examples), 10% (5000 examples) and 20% (10000 examples).

Fig 4.2 shows that SB accelerates training for all three settings. With 1% and 10% of examples corrupted, SB reaches a comparable final test accuracy. With 20% corruption, SB overfits to the incorrect labels and increases the final test error. So, while SB is robust to modest amounts of label error, it is most effective on relatively clean, validated datasets.

Higher selectivity accelerates training, but increases final error. Tuning β in Equation 3.1 changes SB's selectivity. In Figure 3.14, we see that increasing SB's selectivity, focusing more on harder examples, increases the speed of learning but can cause result in higher final error. For CIFAR10, SB reaches within 0.92% of Traditional's final error rate with 20% selectivity. For CIFAR100, it reaches within 2.54% of the final error rate with 25% selectivity. As with other hyperparameters, the best selectivity depends on the target error and dataset. Overall, we observe that SB speeds up training with a range (20–65%) for selectivity.

SB using additional learning rate schedules. We train SB using the provided learning rate schedule in [26] which reproduces state-of-the-art accuracies on CIFAR10, CIFAR100, and SVHN for Wide Resnet with Cutout. We also train using a static learning rate schedule to adjust for confounding factors, as well as an accelerated learning rate schedule. In both cases, we see

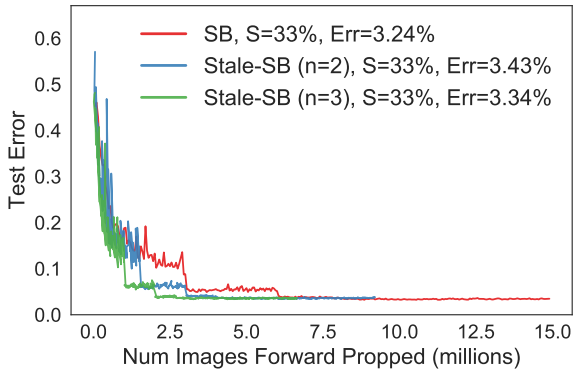
the same trends as with the initial learning rate. We include the configurations in Section 3.6.5 and the training curves in the appendix.

3.6.5 Putting it all together

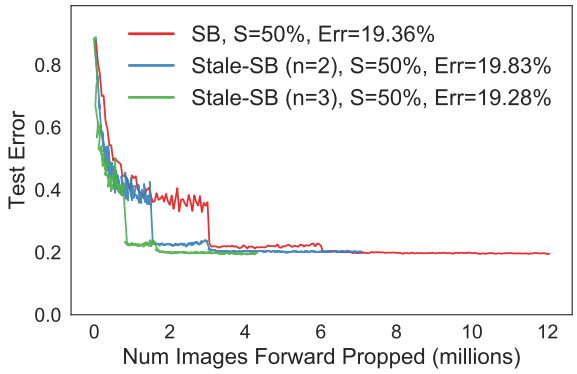
The optimal training setup to reach a certain target error rate depends on a variety of factors. In the previous sections, we compared Traditional, SB, StaleSB, and Kath18 using a variety of different configurations. In Figure 3.15, we plot the wall-clock time needed to reach a range of target error rates for all four strategies, each trained with two learning rate schedules and run with different selectivities. A small subset of configurations make up the Pareto frontier, which represent the best strategy for a given target error rate. Points on the Pareto frontier are colored in bold whereas suboptimal points are shown with transparency.

SB provides the majority of Pareto-optimal configurations. As an approximate signal for robustness of our strategy, we calculate the fraction of Pareto points provided by SB and StaleSB, Kath18 and Traditional. For a majority of training time budgets, SB gives the lowest error rates. For CIFAR10, CIFAR100, and SVHN, SB and its optimized variant StaleSB account for 72%, 47% and 80% of the Pareto-optimal choices, respectively. The exception is cases with very large training time budgets, where Traditional reaches lower final error rates than SB. Overall, Traditional accounts for 10%, 43% and 6% of Pareto points in CIFAR10, CIFAR100 and SVHN, respectively. As shown in Table 3.1, SB is also faster than Kath18, a state-of-the-art importance sampling technique for speeding up training, while achieving the same final error rate. Kath18 provides 10%, 8% and 14% of Pareto-optimal points.

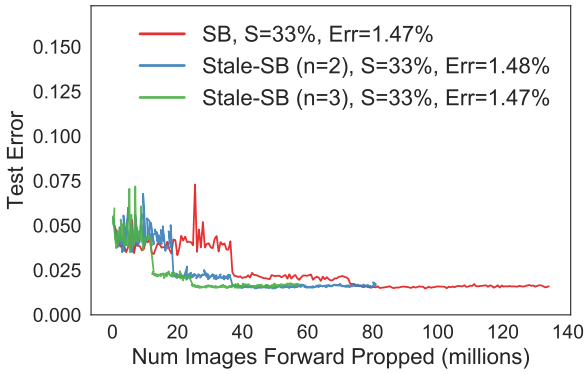
Practicality. Selective-Backprop reduces training iterations and wall-clock time needed to achieve a target error rate with little programmer effort. We evaluated Selective-Backprop with a diverse set of network architectures and datasets. In each case, we did not retune initial hyper-parameters from canonical setups. Most of these training setups included traditional accuracy-boosting techniques, including data augmentation, cutout, dropout, and batch normalization. Selective-Backprop still improved training atop these existing optimizations. Selective-Backprop is also mathematically lightweight and simple to add to code.



(a) CIFAR10

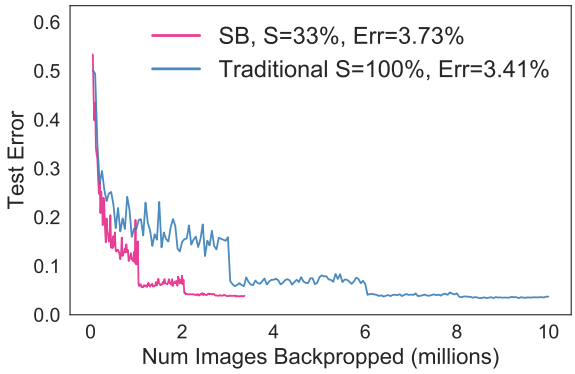


(b) CIFAR100

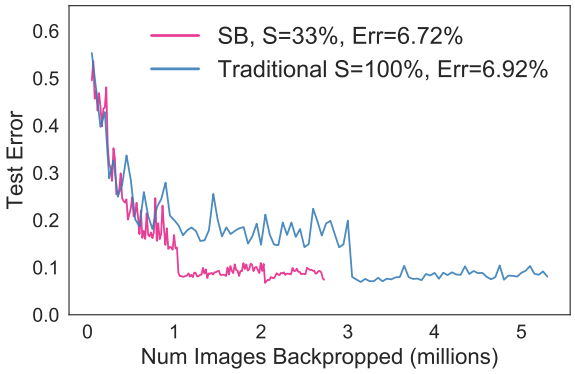


(c) SVHN

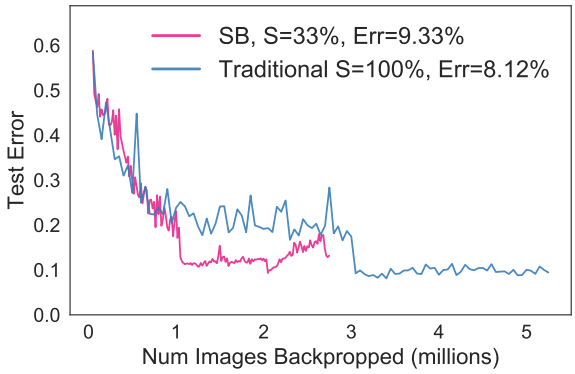
Figure 3.12: Increasing loss staleness reduces number of forward passes with little loss in accuracy.



(a) Test error, 1% shuffled

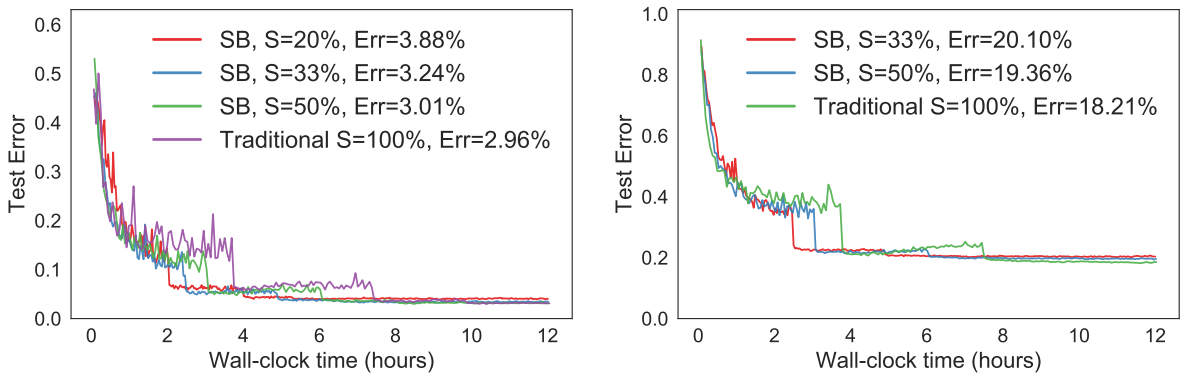


(b) Test error, 10% shuffled



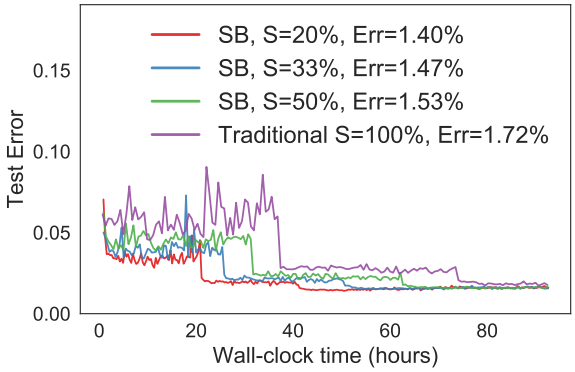
(c) Test error, 20% shuffled

Figure 3.13: SB reaches similar test error rates compared to Traditional with 1% and 10% shuffled labels.



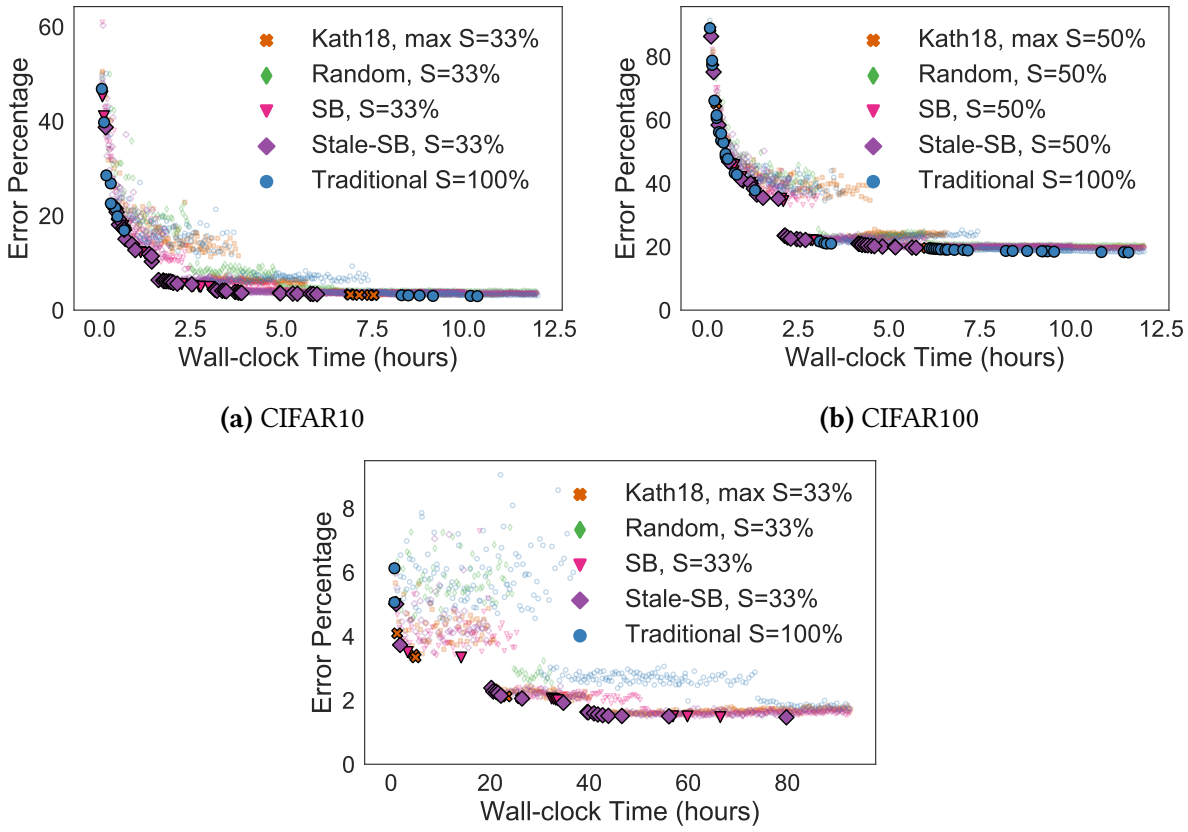
(a) CIFAR10

(b) CIFAR100



(c) SVHN

Figure 3.14: SB accelerates training for a range of selectivities. Higher selectivity gives faster training but can increase error.



(a) CIFAR10

(b) CIFAR100

(c) SVHN.

Figure 3.15: Pareto-optimal points for training time vs error trade-off are opaque and filled. SB and StaleSB offer the majority of Pareto-optimal options for trading off training time and accuracy.

Chapter 4

Adapting Selective-Backprop's Prioritization Function

In this chapter, we address a fundamental limitation of the Selective-Backprop approach presented in the previous chapter. Although Selective-Backprop dynamically determines which examples are challenging, it keeps the mapping of loss percentile to selection probability as static. For example, a training example that exhibits a loss in the 50th percentile of historical losses will always be backpropped 50% of the time. Although this static heuristic used in Selective-Backprop works well on a variety of datasets, it suffers on datasets with label error, often performing worse than traditional training.

The intuition behind the worse performance of Selective-Backprop is that by our current definition of challenging examples (i.e., examples with high loss), mislabeled examples are challenging and thus prioritized. This is a fundamental drawback of static prioritization. In this chapter, we show that dynamically adapting the prioritization function to the dataset increases robustness to label error. In addition, we also adapt the prioritization function during training time and show that doing so improves training sample efficiency.

Our intuition for tuning the degree of prioritization is that the usefulness of challenging examples varies based on the dataset and the time in training. For instance, although prioritizing challenging examples is useful when high-loss examples are informative, it is harmful when these examples are noisy or mislabeled. We test this intuition by tuning Selective-Backprop's degree of prioritization to the dataset and during different times in training, using a variant of Selective-Backprop that we term AdaptSB. AdaptSB generalizes Selective-Backprop by relaxing the assumption that high-loss examples should always be prioritized. Instead, we allow for arbitrary prioritization functions that map an example's relative loss compared to historical examples, to the probability of selecting that example. We find that tailoring this prioritization function to the dataset makes AdaptSB applicable to more datasets than Selective-Backprop (Section 4.3).

Different degrees of example difficulty have different utility at different stages of training. For example, related work in curriculum learning shows that it is useful to first train on examples

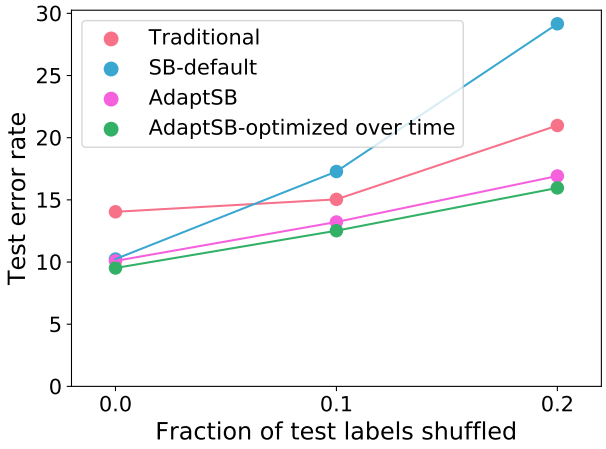


Figure 4.1: AdaptSB accelerates training compared to Selective-Backprop and Traditional by tailoring the prioritization function to the dataset and training time.

that are easy for the network to classify and then gradually introduce harder examples [9]. We explore this idea in AdaptSB: We optimize the prioritization function multiple times over the course of training. We find that dynamically adapting the prioritization function during training time reduces the number of samples needed to reach a target error (Section 4.4).

4.1 Related work

AdaptSB uses importance sampling to bias the training dataset towards useful examples, while de-prioritizing mislabels. We contextualize AdaptSB's approach amongst other importance sampling and re-weighting techniques that mitigate these kinds of dataset bias. AdaptSB also determines what examples are useful at different stages of training, and samples those disproportionately. This idea has been well studied in the literature of curriculum learning. We provide an overview of the work done.

Mitigating training set bias. When using AdaptSB, there exists a tension between two opposing factors. On the one hand, AdaptSB should prioritize high-loss examples to learn from challenging, out-of-distribution examples. On the other hand, high-loss examples could be mislabeled examples that AdaptSB should de-prioritize. This tension also applies to prior loss-based sampling approaches [14, 85, 90, 109]; Selective-Backprop is one such approach. These techniques prioritize examples with larger training losses, increasing their sensitivity to label error [101].

Techniques that tackle the problem of noisy labels [7, 35, 54, 75, 81, 100] do not benefit from prioritizing out-of-distribution examples. Some of these techniques use sampling to prioritize examples with *lower* training losses [81]; others re-weight losses, or change the DNN model to increase its robustness.

AdaptSB balances the presence of *both* out-of-distribution challenging examples and mislabeled examples in the same dataset. Similar to AdaptSB, Ren et al. [101] improve training on datasets with both types of biases, albeit with a different goal than ours. Their approach reweights examples based on their performance on a small, unbiased validation set and aims to improve model generalization. In contrast, in AdaptSB, we use an importance sampling approach with the goal of improving training efficiency.

Curriculum learning. In the field of curriculum learning, DNNs are trained using heuristics designed to emphasize different examples at different times in training. Differing philosophies motivate these approaches, such as supplying easy or canonical examples early in training as in self-paced learning [9, 70]; and emphasizing rare and difficult examples to accelerate learning or to avoid overfitting [6, 54, 60]. Our work takes a data-driven approach instead of applying a heuristic, by testing the usefulness of examples of different degrees of difficulty with a hyperparameter search.

4.2 AdaptSB

AdaptSB is a variant of Selective-Backprop that dynamically prioritizes different example difficulties when encountering different datasets and stages of training. For example, challenging examples may be useful when training on a sanitized dataset, but could be detrimental when they include mislabeled examples.

4.2.1 Prioritization functions

We control the relative prioritization of examples with differing degrees of difficulty using a prioritization function, $\mathcal{F}$. A prioritization function maps the relative loss of an example to the probability of selection for training. Formally, we define it as:

$$\mathcal{P}\Big(\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)\Big) = \mathcal{F}\Big[\text{CDF}_R\Big(\mathcal{L}(f_{\mathbf{w}}(\mathbf{x}_i), y_i)\Big)\Big]$$

CDF_R is a running tally of the losses of the last R examples and $\mathcal{P}$ is the probability that example i is selected for backprop. $\mathcal{L}$ is our loss function used to compare the output of model $f_{\mathbf{w}}$ on example $\mathbf{x}_i$ and the ground truth, y_i. $\mathcal{F}$ maps from [0,1] to [0,1]. In Selective-Backprop, we chose $\mathcal{F}$ to be an exponential transform so that low values of CDF_R have a low probability of selection. In AdaptSB, we relax this constraint, and optimize $\mathcal{F}$ to accelerate training on a given dataset or stage of training.

4.2.2 Optimizing the prioritization function

We search the space of possible $\mathcal{F}$s with a hyperparameter search. The goal is to find a function that minimizes the final error rate given a training time budget. During the search, a Gaussian process algorithm proposes candidate functions. Compared to using a random search algorithm, we found that a Gaussian process was more efficient at finding functions that minimized final error. AdaptSB uses the prioritization for training for a time budget and reports the final error to the search algorithm. These results are used to inform the next candidate prioritization until the search is completed.

We design the structure of $\mathcal{F}$ to be expressive but also tractable for our hyperparameter search. For instance, the exponential transforms used in Selective-Backprop are not expressive enough to represent prioritization functions that deemphasize the most challenging examples. On the other hand, a function where each Y-value is independently determined may be representative, but not tractable to optimize. We use a function that falls in the middle.

We represent $\mathcal{F}$ with a linear interpolation of a fixed number of points, where the X-values are set in advance and the Y-values are experimentally determined. In our experiments, we found that a linear interpolation performed similarly to a cubic interpolation. We used four points, because doing so allowed our optimization process to converge after trying $\approx$300 proposed $\mathcal{F}$s. However, more points can be used if the computational resources and search time budget allow. We chose to fix our X-values to 0, 0.5, 0.8, and 1. Intuitively, the first Y-coordinate dictates the probability of selection for the example with the lowest loss. The second Y-coordinate dictates the selection probability for the example with the median loss. We chose these X-values as it allowed us to map all the points between 0 and 1, while focusing our degrees of freedom towards more challenging examples. Figure 4.3 shows an example prioritization function. We found that with this setup, optimizing $\mathcal{F}$ is both tractable and improves training (Section 4.3 and 4.4). However, there are possible alternatives to $\mathcal{F}$ that may reduce the search time and further improve training. We discuss some of these potential functions in Section 4.5.

4.2.3 AdaptSB end-to-end

We believe AdaptSB could be used for training end-to-end, as a replacement for Selective-Backprop. AdaptSB has the same online training process as SB. It uses an example's loss to determine its likelihood of being sampled for training. However, while Selective-Backprop prioritizes high loss examples, AdaptSB searches for and uses a tailored prioritization function, $\mathcal{F}$. Compared to SB, AdaptSB requires an additional step of searching for a prioritization function that minimizes error given a training time budget. In order to realize the end-to-end benefit of AdaptSB, we need to minimize the relative cost of its hyperparameter search.

If AdaptSB's prioritization function is optimized once per dataset, the hyperparameter search can be run offline before training starts. AdaptSB uses the training dataset (or a subset), and trains

with different prioritization functions. The best performing configuration replaces the default prioritization function and training then proceeds identically to that of Selective-Backprop.

AdaptSB can also adapt $\mathcal{F}$ over the course of training. In this case, the hyperparameter search should be performed asynchronously and in parallel to training. That way, AdaptSB can find new prioritization functions without disrupting training. Once the search is complete, the system will dynamically update its prioritization function to the latest found configuration. In order to run the search asynchronously, AdaptSB uses a snapshot of the model. Since training doesn't stop for the hyperparameter search, the model used for optimization is stale. One way to minimize staleness is to complete the hyperparameter search faster. However, both model staleness and reducing the search time budget could adversely impact the final result. A future research direction is to determine the trade-off between stopping the hyperparameter search early and reducing model staleness.

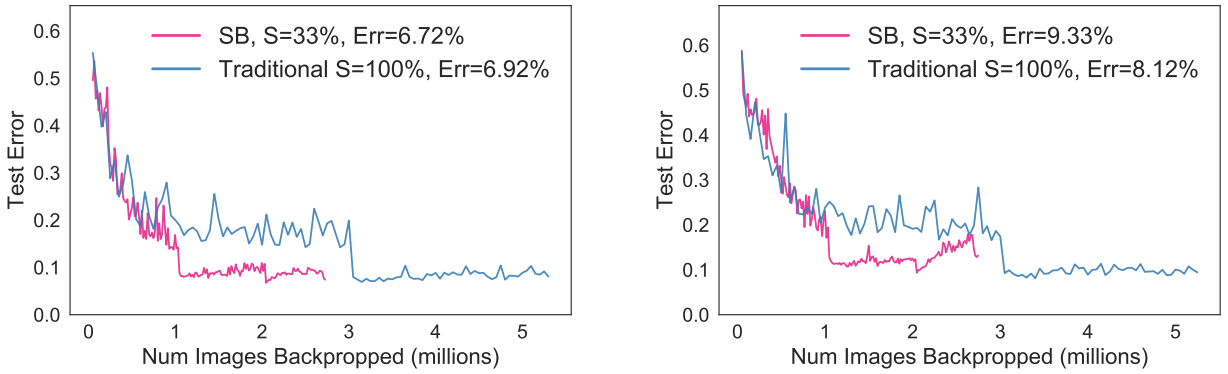
4.3 Evaluation: Dynamically adapting to datasets

By tailoring the prioritization function to the dataset, AdaptSB accelerates training on datasets that Selective-Backprop does not perform well on. In particular, we find that Selective-Backprop struggles to train models on datasets with label error because mislabeled examples, which generate high loss, are over-emphasized during training. For these datasets, AdaptSB learns to ignore mislabels and trains more efficiently than Selective-Backprop and traditional training with SGD.

Selective-Backprop is robust to small amounts of label error. We observe that Selective-Backprop can increase susceptibility to noisy labels. On SVHN, a dataset known to include label error [95], Selective-Backprop still converges faster than Traditional to almost all target error rates (Chapter 3). In this section, we evaluate Selective-Backprop on CIFAR10 with manually corrupted labels. Following the UniformFlip approach in [101], we randomly flip 1% (500 examples), 10% (5000 examples) and 20% (10000 examples). Selective-Backprop accelerates training for all three settings. While with 10% of examples corrupted, Selective-Backprop reaches a comparable final test accuracy, with 20% corruption, Selective-Backprop overfits to the incorrect labels and increases the final test error (Figure 4.2). So, while Selective-Backprop is robust to modest amounts of label error, it is most effective on relatively clean, validated datasets.

AdaptSB is more robust to label error. We use AdaptSB to train on CIFAR10 and CIFAR10 with injected mislabels. For each dataset, we run a hyperparameter search using a hold-out validation set. The search tries 300 potential configurations, training for two hours for each configuration. We use a Gaussian process algorithm to find configurations that maximize the average accuracy of the last three epochs of training. The search took 40 hours to complete on 15 machines.

The best prioritization function found by AdaptSB de-prioritizes challenging examples for datasets with mislabels (Figure 4.3). For CIFAR10 with 0% label error, AdaptSB prioritizes the challenging examples that represent the 80th percentile of losses or higher. (Interestingly, it pri-



(a) Test error, 10% shuffled

(b) Test error, 20% shuffled

Figure 4.2: Training diverges with Selective-Backprop on datasets with large amounts of label error.

oritizes the easiest examples over the 50th percentile of examples. We have not yet diagnosed why this is the case.) When 10% of labels are randomized, AdaptSB de-prioritizes the most challenging examples, only using them 40% of the time. When 20% of labels are randomized, the most challenging examples become the least used, only being selected for training less than 20% of the time. We validate AdaptSB's efficacy by training using the chosen configurations for two hours on a different test set. On CIFAR10, AdaptSB achieves 0.15%, 4.06%, and 12.24% lower error compared to Selective-Backprop with 0%, 10%, and 20% of test labels randomized (Figure 4.1), respectively. While Selective-Backprop performs worse than Traditional on datasets with label error, AdaptSB achieves a 1.82% and 4.06% lower error rate when 10% and 20% of labels are shuffled, respectively.

While our experiments use randomly-injected label error, in real datasets with label error, these mislabels are likely to be correlated. For example, it is more likely to mislabel a plane as a ship than as a dog. While we believe that datasets with injected label error is a good starting point to understanding the effect of increasingly more mislabels on training, an interesting direction for future work is to analyse AdaptSB on datasets with true label error.

4.4 Evaluation: Dynamically adapting during training

We next apply AdaptSB's hyperparameter search at different stages of training to determine how the optimal prioritization function changes over time. This is analogous to other adaptive hyperparamater schedulers (e.g., for setting the learning rate), which tune hyperparameters based on the current state of training. We call this approach AdaptSB-Multi. Our analysis of AdaptSB-Multi further supports the hypothesis of curriculum learning that different examples are useful at different stages of training. We make the following observations.

High-loss examples are more useful later in training. First, we train CIFAR10 using the setup specified in Section 3.6. To observe which examples are the most sample-efficient during

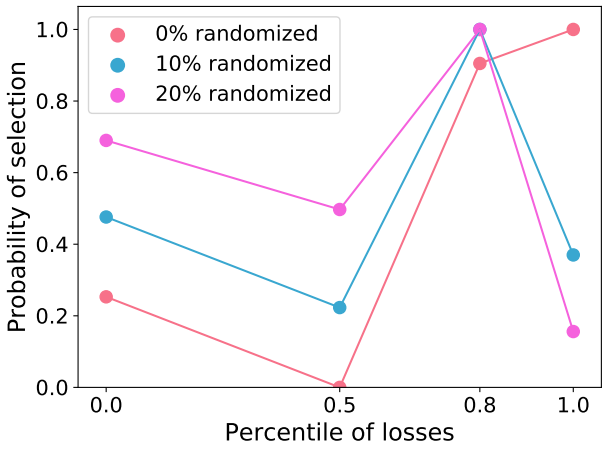


Figure 4.3: The prioritization function AdaptSB finds for each dataset. AdaptSB deemphasizes hard examples as more mislabels are introduced into CIFAR10.

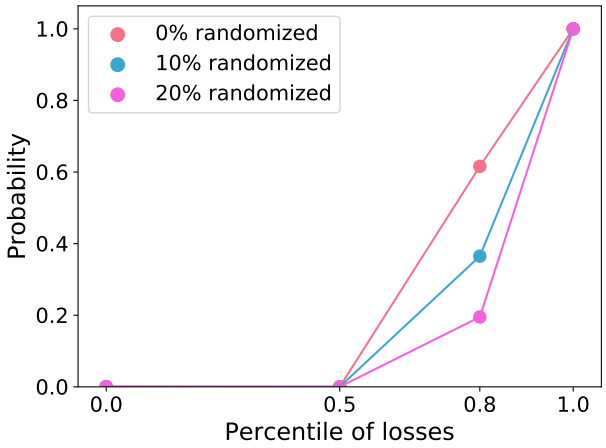


Figure 4.4: The prioritization function AdaptSB finds for each stage of training. AdaptSB emphasizes hard examples as training progresses.

different stages of training, we train for 0 epochs, 30 epochs, 70 epochs, and 160 epochs using Traditional and tune the prioritization function using the resulting models. Unlike Bengio et al. [9], we do not find that easy examples should be emphasized early in training. Even at the start of training, AdaptSB prioritizes challenging examples over easy examples. The examples that result in the lowest percentile of loss are backpropped less than 25% of the time. However, our results support Bengio et al.'s hypothesis that high-loss examples should be increasingly prioritized as training proceeds. After 30 epochs of training, AdaptSB ignores examples that result in the lowest 50% of losses altogether (Figure 4.4).

AdaptSB-Multi finds sample-efficient examples. To put it all together, in AdaptSB-Multi, we train using AdaptSB and continuously update the prioritization function by running additional hyperparameter searches during training. Since a hyperparameter search is relatively costly, there is a tradeoff between performance and the frequency of recomputing the prioritization function. As an initial experiment, we train using a time budget for two hours and recompute the prioritization curve three times: before training, after 40 minutes of training, and after 80 minutes of training. We run the hyperparameter searches on a holdout validation set and measure the results on the test set. For CIFAR10 with 0%, 10% and 20% labels shuffled, AdaptSB-Multi reaches a 0.6%, 0.7%, and 1% lower test error in absolute terms than AdaptSB (Figure 4.1).

4.5 Future work: Towards a wall-clock speedup

In the previous sections, we have shown that the prioritization function configurations proposed by AdaptSB and AdaptSB-Multi allow us to reach lower target errors for the same training time budget. However, to achieve end-to-end wall-clock speedups, the time spent in AdaptSB's hyperparameter search must be taken into account. We believe that the hyperparameter search can be small relative to the overall training time. There are two ways to achieve this. First, when using AdaptSB on a very large dataset, such as Google's internal JFT dataset which includes over 300 million training examples, the relative cost of a hyperparameter becomes smaller. AdaptSB can be run on a small holdout dataset which is representative of the rest of the dataset. Second, there are potential optimizations to AdaptSB that would reduce the absolute value of the time taken. We save trying and evaluating these for future work but list them here for completeness.

4.5.1 Reducing the search space

One approach to reducing the hyperparameter search time is to reduce the search space. We propose three possible techniques for doing so.

Discretize the possible hyperparameter values. Currently, AdaptSB searches for four hyperparameters, representing the Y-values of four static X-axis points. Each value is a probability, and is therefore constrained as a real-value between 0 and 1. However, in order to reduce the search space, we could quantize the possible values, e.g., to be one of ten values between 0 and

1. We expect this to reduce the number of trials needed, as currently many of the later trials of the hyperparameter search are spent tweaking one or two parameters by less than 0.1. However, this may sacrifice the final accuracy improvement gained by this fine-tuning. The efficacy of this approach therefore depends on the sensitivity of AdaptSB and the dataset used.

Use domain-specific knowledge. Another approach to constraining the search space is to employ domain-specific knowledge about our problem. For instance, we could assume that at least one of the prioritization function Y values should be equal to 1. This is because we are interested in finding the relative priority between different types of examples. We can always normalize the highest priority bucket to have a probability of 1. Another approach is that if we knew a dataset was well-sanitized (e.g., CIFAR10), we could assume that high-loss examples should be prioritized. We could use AdaptSB to determine the correct intensity of prioritization by constraining the prioritization function to monotonically increase.

Use a different prioritization function. In AdaptSB, we chose to represent our prioritization function with four points splined together as it is simple and easy to interpret. This function abstraction requires a hyperparameter search for four free hyperparameters. Instead, we can reduce the search space by using a prioritization function definition which has fewer free parameters. For example, the beta distribution uses only two hyperparameters but can be even more expressive.

4.5.2 Faster search

In addition to reducing the size of the search space, we also suggest approaches to improve search efficiency.

Sensitivity analysis. We can conduct sensitivity analyses on the effect of small changes in hyperparameter values, in order to understand the need for many trials or longer training time budgets. For instance, we can experimentally search for the minimum number of trials in a hyperparameter search needed to reach 99% of the accuracy gains. We can also minimize the training time used to determine the efficacy of each potential configuration.

Hyperparameter search algorithms. AdaptSB uses the Gaussian process algorithm, implemented in Sherpa [28]. We can explore other search algorithms or hyperparameter tuners. For instance, we could use Hyperband [74], which employs early stopping and is therefore likely able to tune the prioritization function more efficiently.

4.6 Conclusion

AdaptSB delivers Selective-Backprop's benefits with less sensitivity to label error. It dynamically chooses which degrees of example difficulty to prioritize. On datasets with label error, AdaptSB de-prioritizes challenging examples and trains faster than traditional training and Selective-Backprop.

On CIFAR10 with 20% of training labels shuffled, AdaptSB decreases the error rate by 12.24% compared to Selective-Backprop. AdaptSB-Multi further improves sample efficiency by increasing the emphasis on challenging examples over the course of training. Using AdaptSB-Multi over time on CIFAR10 with 20% of training labels shuffled further decreases the error rate by 1%. This provides evidence to the benefits of dynamically adapting the prioritization of different example difficulties to the dataset and stage of training.

Chapter 5

Adaptive Stem-Sharing for Multi-Tenant Video Processing

5.1 Overview

Mainstream is a new video analysis system that jointly adapts concurrent applications sharing fixed edge resources to maximize aggregate result quality. Mainstream exploits partial-DNN (deep neural network) compute sharing among applications trained through transfer learning from a common base DNN model, decreasing aggregate per-frame compute time. Based on the available resources and mix of applications running on an edge node, Mainstream automatically determines at deployment time the right trade-off between using more specialized DNNs to improve per-frame accuracy, and keeping more of the unspecialized base model to increase sharing and process more frames per second. Experiments with several datasets and event detection tasks on an edge node confirm that Mainstream improves mean event detection F1-scores by up to 47% relative to a static approach of retraining only the last DNN layer and sharing all others ("Max-Sharing") and by 87X relative to the common approach of using fully independent per-application DNNs ("No-Sharing").

Video cameras are ubiquitous, and their outputs are increasingly analyzed by sophisticated, online deep neural network (DNN) inference-based applications. The ever-growing capabilities of video and image analysis techniques create new possibilities for what may be gleaned from any given video stream. Consequently, most raw video streams will be processed by multiple analysis pipelines. For example, a parking lot camera might be used by three different applications: reporting open parking spots, tracking each car's parking duration for billing, and recording any fender benders.

Mainstream focuses on improving video processing on edge devices, which will be a common way to address bandwidth limitations, intermittent connectivity (e.g., in drones), and real-time requirements. Applications executing at the edge, though, face tighter bounds on resource availability than in datacenters. Naturally, optimal video application performance requires tuning for

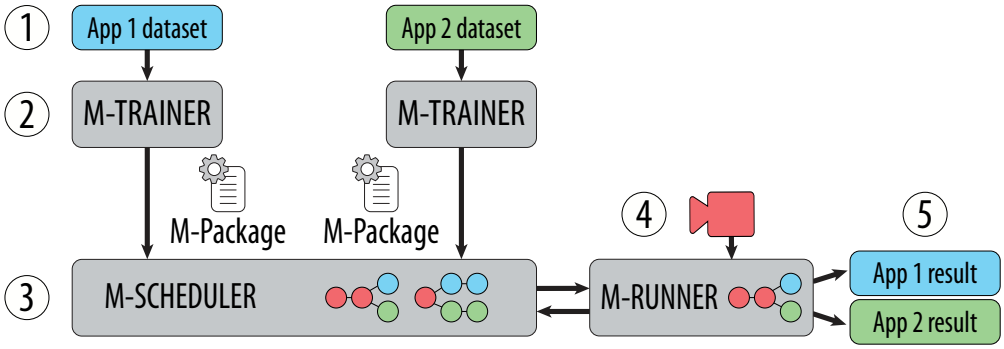


Figure 5.1: Mainstream Architecture. Offline, for each task, M-Trainer takes a labeled dataset and outputs an M-Package. M-Scheduler takes independently generated M-Packages, and chooses the task-specific degree of specialization and frame rate. M-Scheduler deploys the unified multi-task model to M-Runner, performing inference on the edge.

the resources available [21, 40, 59, 125, 129].

Unfortunately, what resources will be available to the application at deployment time is often unknown to the developer. Further, resource availability changes as additional applications arrive and depart. Instead, individual application developers typically develop their models in isolation, assuming either infinite resources or a predetermined resource allotment. When a number of separately tuned models are run concurrently, resource competition forces the video stream to be analyzed at a lower frame rate—leading to unsatisfactory results for the running applications, as frames are dropped and events in those frames are missed. However, due to the popularity of *transfer learning* (Sec. 2) [94, 98, 110, 123], contention can be reduced by eliminating redundant computation between concurrent applications [40].

Mainstream is a new system for video processing that addresses resource contention by dynamically tuning degrees of work sharing among concurrent applications. Specifically, it focuses on sharing portions of DNN inference, which consumes the majority of video processing cycles. Mainstream exploits the potential "shared stem" of computation that results from application developers' use of the standard DNN training approach of transfer learning. In transfer learning, training begins with an existing, *pre-trained* DNN, which is then re-trained for a different task. Typically, only a subset of the pre-trained DNN is specialized; when different applications start with the same pre-trained DNN, Mainstream identifies the common layers and executes them only once per frame.

A critical challenge of exploiting shared stems well is determining how much to share. Application developers usually specialize as much of the pre-trained DNN as is necessary to achieve high model accuracy. More specialization, however, means that less of the pre-trained DNN can be shared. Thus, there is an explicit trade-off between the benefits of greater per-frame accuracy (via more-specialized DNNs) and processing more frames of the input video stream (via more sharing of less-specialized DNNs). The right choice depends on the edge device resources, the number of concurrent applications, and their individual characteristics.

Deployment time model selection. Mainstream moves the final DNN model selection step from application development time to deployment time, when the hardware resources and concurrent application mix are known. By doing so, Mainstream has the necessary information to select the right amount of DNN specialization (and thus sharing) for each application. As applications come and go, Mainstream can dynamically modify its choices. Previous systems like VideoStorm [125] select models by considering each application independently. *The specialization-vs-sharing trade-off, however, can only be optimized when considering applications jointly.* Joint optimization produces a combinatorial set of options, which Mainstream navigates using application metadata and domain-specific models; the system uses this information to estimate the effects of DNN specialization and frame sampling rate on application performance. Unlike black-box approaches, Mainstream can jointly optimize for stem-sharing without needing to profile each combination.

Mainstream consists of three main parts (Figure 5.1). The *M-Trainer* toolkit helps application developers manage their training process to produce the information needed to allow tuning the degree of specialization at deployment time. Current standard practice is for developers to experiment with different model types, hyperparameters, and degrees of re-training to find the best choice for an assumed resource allocation, discarding the trained DNN models not chosen. M-Trainer instead keeps "less optimal" *candidate* DNN models, together with associated training-time information (e.g., per-frame accuracy, event detection window). The *M-Scheduler* uses this information, together with a profile of per-layer runtime on the target edge device, to determine the best candidate for each application—including the degrees of specialization and, thus, sharing. It efficiently searches the option space to maximize application quality (e.g., average F1 score among the applications). The *M-Runner* runtime system runs the selected DNNs, sharing the identical unspecialized layers.

Experiments with several datasets and event detection tasks on an edge node confirm the importance of making deployment-time decisions and the effectiveness of Mainstream's approach. Results show that dynamic selection of shared stems can improve F1-scores by up to 87X relative to the common approach of using fully-independent per-application DNNs (No-Sharing) and up to 47% compared to a static approach of retraining only the last DNN layer and sharing all others (Max-Sharing). Across a range of concurrent applications, Mainstream adaptively selects a balance between per-frame accuracy and frame sampling rate that consistently provides superior performance over such static approaches.

Contributions. This chapter makes three main contributions. First, it highlights the critical importance of reducing aggregate per-frame CPU work of multiple independently developed video processing applications via stem-sharing; No-Sharing is unable to support even three concurrent applications on our edge node deployment. Second, it identifies the goodness trade-off between per-frame quality and the frame sampling rate dictated by the degree of DNN specialization (and thus the amount of sharing). Third, it describes and demonstrates the efficacy of the Mainstream approach for automatically deploying the right degree of specialization for each submitted application's DNN.

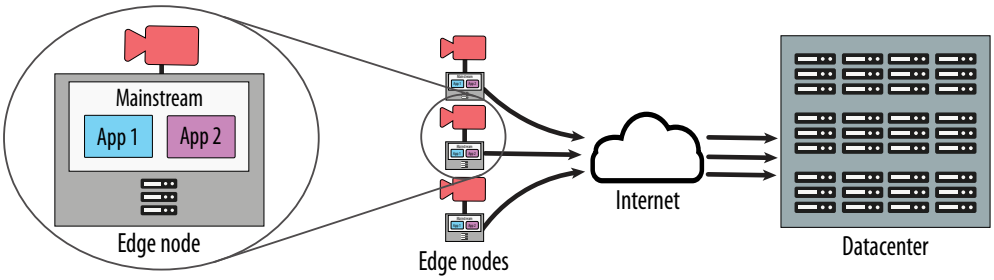
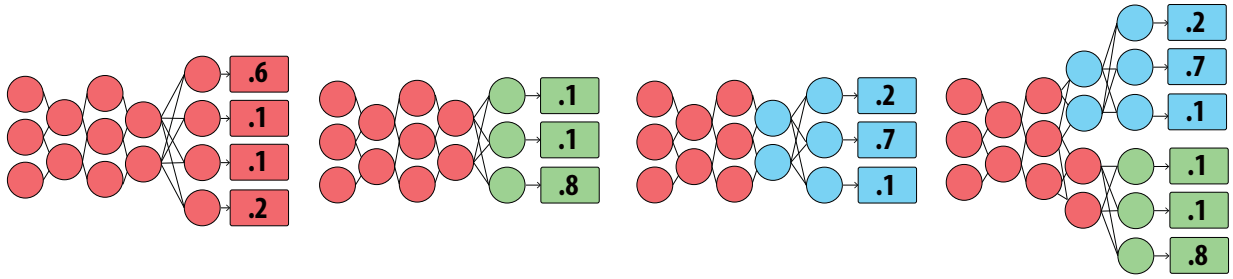


Figure 5.2: Overview of environment Mainstream runs in.

In the context of Mainstream, we assume a live analytics platform (Figure 5.2), wherein embedded processing is co-located with each camera to analyze the raw video and send out results. A key issue with such a system is that a fixed, limited amount of computing capacity is available per camera stream; processing can not scale up and down as can happen in a shared data center. Furthermore, as it is impractical to frequently upgrade the deployed infrastructure, such platforms may remain in service for many years. So, the average embedded compute will likely be older and less powerful hardware. Thus, analytics must be made to fit and should use the full resource. Furthermore, since new applications will be developed over the life of a given platform, the system must efficiently handle multiple, concurrent, independently-developed analytics applications. We describe how Mainstream does so via adaptive sharing of computation among DNN-based analytics applications.

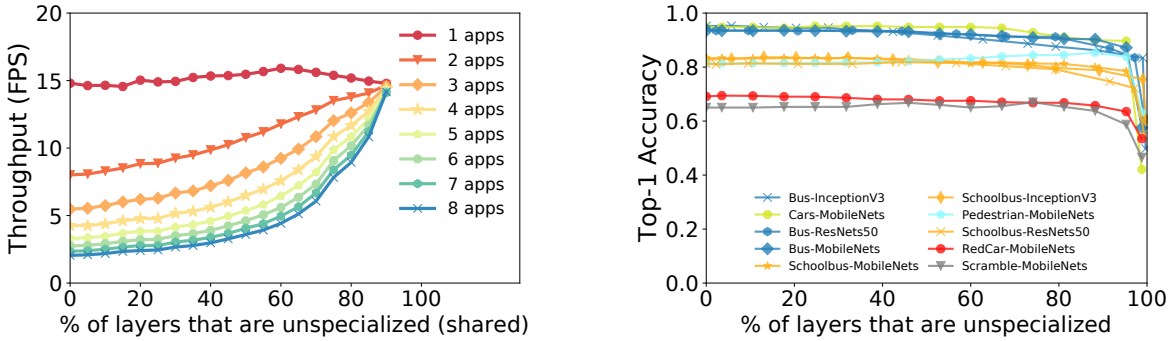


(a) Base DNN model **(b)** App. #1's DNN model **(c)** App. #2's DNN model **(d)** As executed in MS

Figure 5.3: Figure 5.3a depicts a base DNN trained from scratch for its task. Figure 5.3b and Figure 5.3c show two new task DNNs, fine-tuned against the base DNN. App. #1 freezes more layers during training than App. #2. Figure 5.3d shows how Mainstream runs the applications concurrently. Layers frozen by both App. #1 and App. #2 can be shared.

5.2 Mainstream approach

Finetuning or transfer learning is an alternative to training a model from scratch where weights are initialized randomly. Here, a model that has already been trained on a similar task (a *base*



(a) Throughput vs. Sharing

(b) Per-frame accuracy vs. (Potential) sharing

Figure 5.4: Conflicting consequences of DNN compute sharing. (a) shows the frame processing rate for 1–8 concurrent event detection applications as a function of the fraction of the InceptionV3 DNN they share, from No-Sharing on the left to sharing all but the last layer (Max-Sharing) on the right. The experiments are run on the hardware described in Section 5.4. (b) shows Top-1 accuracy as a function of the fraction of unspecialized layers for three popular DNN architectures (ResNet-50 [42], InceptionV3 [114], and MobileNets-224 [46]) using six of the datasets described in Table 5.1. We trained each classifier using all three network architectures but omitted some curves for brevity. The horizontal axis starts from fully specialized DNNs on the left to only the last layer being specialized on the right; recall that potential computation sharing is limited to the unspecialized layers.

DNN as in Figurere 5.3a), is used as an initialization point or feature extractor for the new *target DNN*. During training, a subset of the old parameters are *frozen* and do not change. The remaining free parameters are then retrained on the new task with a new training dataset (Figurere 5.3b and Figurere 5.3c). This process *fine-tunes* these parameters to achieve a result comparable to end-to-end training on the entire DNN, but does so with much less data and at a much lower computational cost. In practice, few practitioners train networks from scratch, let alone develop novel network architectures. Transfer learning via one of a few popular neural networks is standard practice. This work leverages the fact that many deployed DNNs will use transfer learning to adapt a handful of existing, effective, published models to perform new related tasks. As we will illustrate, the common structure and related origins between multiple different DNNs will enable computation sharing between these tasks, allowing very efficient execution on limited edge computing resources.

Sharing computation between DNNs. When supporting multiple inference applications on a single infrastructure, the common approach is to execute every application's DNN model independently. We refer to this as a "No Sharing" approach. To avoid redundant work, Mainstream instead computes results for DNN layers common to multiple concurrent applications just once and distributes the outputs of shared stems to the specialized layers of all sharing applications. This is analogous to common subexpression elimination used in other domains, e.g., optimizing compilers or database query planners.

Figure 5.3 illustrates how compute sharing can be leveraged when two DNNs are fine-tuned

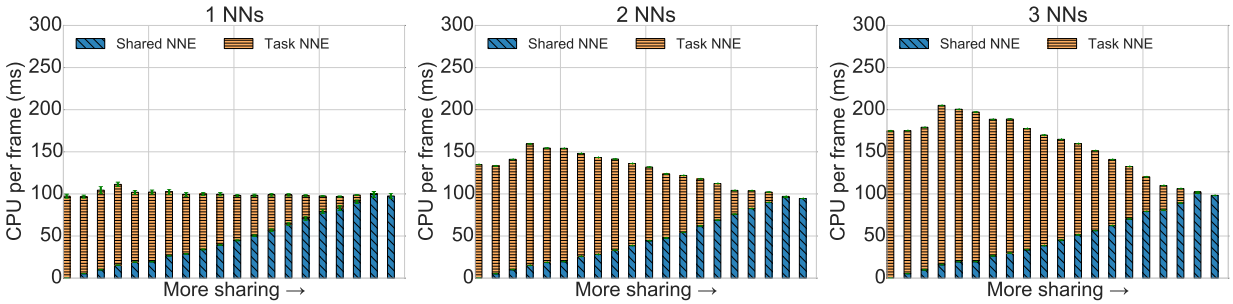


Figure 5.5: Latency of shared and task-specific DNNs using different amounts of sharing between applications. There is a larger latency when we share and transfer the output of convolution layers with large intermediate activations.

from a common pre-trained model and have some unspecialized layers in common. Compute sharing can significantly affect per-frame computation cost and improve throughput for a given CPU resource. Figure 5.4a quantifies this effect. It shows the throughput achieved by Mainstream running up to eight concurrent InceptionV3-based event detection pipelines, as a function of how many DNN layers they have in common (i.e., their common degree of specialization). With no sharing (the left-most points), adding a second application halves throughput, which continues to degrade geometrically as more applications are added. Moving to the right, throughput improves as more layers are shared. When all but the last layer are shared, additional applications can be run at very low marginal cost.

On the other hand, there are costs to enabling sharing by leaving many layers unspecialized. In particular, the per-frame accuracy of a model may be lower when only a few layers are specialized. Figure 5.4b shows the effect of specialization on per-frame accuracy for several combinations of DNN architectures and classification tasks. As expected, accuracy decreases slowly as less-specialized networks are employed (and hence more sharing is enabled)— with a large decrease occurring only when the fraction of the network specialized is very small. This characteristic enables Mainstream to share large portions of the network with low accuracy loss.

Throughput and latency. Mainstream scales in terms of the number of applications that can be run on resource constrained nodes. Figure 5.4a shows the throughput achieved running up to eight tasks. The X-axis represents increasing amounts of sharing between applications from deploying a full neural network for each task to sharing all but the last layer. Without sharing, adding a second application halves throughput and additional applications continue degrading throughput geometrically. Performance improves as we share more layers. When we share all but the last layer, additional applications can be run at almost no cost.

Model accuracy. Mainstream's extreme scalability comes at the potential cost of some reduction classification accuracy. Figure 5.4b illustrates this cost as we freeze more NN-cells during training. We show the image-level accuracy of two classification tasks on three DNN architectures. In practice, it is common to fine-tune only the last layer of a task-specific neural network.

This indicates that, in the real world, the benefits of minimal fine-tuning often outweigh the accuracy penalty. At the opposite extreme, freezing just a couple of NN-cells significantly boosts performance with minimal accuracy loss. As we've demonstrated, many of the points in between are Pareto optimal, and we will show that they are often optimal configurations for maximizing user's objectives.

Adaptive management of the sharing opportunity. Since transfer learning is so commonly used by ML developers, and base models are shared within organizations and on the Internet, there may be many opportunities to exploit inter-DNN redundancy in the unspecialized layers. Most developers either use a popular default of specializing only the last layer (which is great for sharing potential, at the potential cost of model accuracy) or determine the degree of specialization based on the amount of training data available, since retraining too many layers without sufficient training data leads to over-fitting. Notably, each developer decides independently.

The problem with this approach is that the impact of sharing computation on application quality depends on factors only known at deployment time: the set of concurrent applications and the resources of the edge node on which they are run. Hence, Mainstream defers the decision regarding how much specialization to employ from application development time to deployment time.

Impact of sampling rate for event detection. Given the trade-off between per-frame accuracy and frame processing throughput, picking the right degree of specialization is challenging. Consider an application for monitoring environmental pollution from trains, which is being built using a train detector we deployed. When the application detects a train coming into view, it triggers the capture of high fidelity frames of the train's smoke stack (for subsequent pollution analysis).

Increasing specialization to improve per-frame accuracy increases the probability of correctly classifying frames containing trains— but reduces shared computation. This, in turn, leads to less frequent sampling, which removes opportunities to analyze frames containing a particular view of a train. A higher frame rate increases the probability that an event will be observed in more frames, creating more opportunities for detection. So, the question becomes: should one sample more frames using a less accurate model or sample fewer frames using a more accurate model?

Analytical model for event detection. The Mainstream scheduler (Sec. 5.3.3) navigates this "accuracy vs. sampling rate" space by evaluating various candidate *{specialization, frame rate}* tuples. To do so, however, the scheduler must be able to interpret the benefit at the application (not per-frame) level. Hence, we propose an analytical model (sketched in equations below) that approximates the *event* F1-score for a given DNN, given estimates of (a) event length, (b) event frequency, (c) the correlation between frames (discussed below), (d) per-frame DNN accuracy, and (e) DNN analysis frame rate. The frame rate (e) comes directly from the scheduler's proposed tuple; similarly, the accuracy (d) associated with a given specialization proposal will be available to the scheduler (see Sec. 5.3.1). Values for event length (a), frequency (b), and correlation (c) can either be measured using representative video samples, or they can be estimated by the developer.

We are able to predict the application's F1-score by estimating the expected number of frames per event that we will have the opportunity to analyze and computing the probability that analyzing the set of frames will result in a detection. The expected number of frames analyzed per event is dependent on the event length and frame rate. The per-frame Top-1 accuracy represents the probability that we will classify any individual frame correctly. However, this does not factor in the fact that sequential frames of an event may be correlated in some way. We therefore introduce and estimate the *inter-frame correlation*, which measures the marginal benefit of analyzing more frames of a single event.

The inter-frame correlation, *corr*, is based on conditional probabilities. It can be measured empirically by looking at the subset of cases where the event wasn't detected in the first frame, and determining what percentage of those cases also didn't detect the event in the second.

We first introduce some formalisms to model this issue, where X and Y are consecutive frames.

$$
\begin{aligned}
P(X) &= \text{Chance of detecting event in frame } X \\
P(\neg X) &= \text{Chance of not detecting event in frame } X \\
P(Y) &= \text{Chance of detecting event in frame } Y \\
P(\neg Y) &= \text{Chance of not detecting event in frame } Y \\
P(\neg Y \& \neg X) &= \text{Chance of not detecting event in either frame}
\end{aligned}
$$

Using the Kolmogorov definition, we can calculate the probability we miss the event in both frame X and Y, with $P(\neg Y \& \neg X) = P(\neg Y \mid \neg X) * P(\neg X)$. Finally, the chance we detect the event in at least one frame becomes the complement: $1 - P(\neg Y \mid \neg X) * P(\neg X)$.

We use the event length and sample rate to derive how many frames of the event we encounter in expectation. Knowing the number of frames analyzed, N, and the probability of detecting an event given N tries, we are able to calculate the expected false negative rate.

Discussed earlier, the model accuracy and event length can come from application definition and the training process (see, e.g., Figure 5.4b), whereas the frame sampling rate depends on the set of concurrent applications and the edge hardware (see, e.g., Figure 5.4a). Given video clips of event examples at training time, the inter-frame correlation can also be computed.

We use Equation 1 to calculate N, the expected number of frames of the event that the model will process. Here, d is the event length, and $stride$ is the inverse of the frame rate. Equation 3 estimates the probability the DNN misclassifies all N analyzed frames. Recall is the complement: the probability that we correctly classify at least one frame of the event.

To estimate the false positive rate, we repeat this calculation, except that d is the number of frames between events (derived from the event frequency), and P_{miss_1} is the probability of *true* negatives. The true positive rate, the false positive rate, and the event frequency are used to calculate the precision. The F1-score is the harmonic mean between precision and recall.

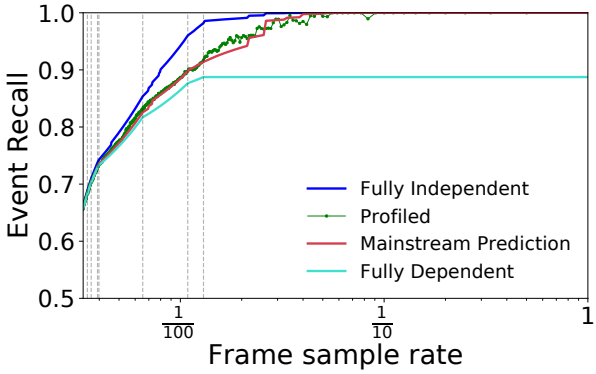


Figure 5.6: Effect of sample-rate on recall, for different inter-frame correlations. The dotted vertical lines represent each train in the dataset, denoting $\frac{1}{trainlength}$, which is the sample rate required to ensure that one frame of that train is analyzed. The "Profiled" line is measured directly from the TRAIN video dataset, and the other three are approximations based on different inter-frame correlations (uncorrelated, fully correlated, and the empirical correlation observed in the TRAIN dataset).

To evaluate our model, we profile an application and measure the actual recall observed when running the event detector application (e.g., train detection) on the video stream at different sampling rates. Figure 5.6 shows observations of how the sample-rate affects the recall of our TRAIN detection task. The train detector is run on a video dataset we collected of train tracks that is described in Table 5.1. The "Profiled" line shows the actual recall observed when running the train detector on the video stream at different sampling rates. The remaining three lines are calculated using our analytical model.

The independent, dependent and correlation lines are calculated as described above, based on assumed assumed conditional probabilities (1, 0, and 0.17, which is the average value observed empirically). The dotted vertical lines represent each train in the dataset, denoting $\frac{1}{trainlength}$, which is the sample rate required to ensure that one frame of the train is analyzed.

In the fully independent (uncorrelated) case, the false negative rate continues to decrease as the sample rate increases. In the fully dependent (correlated) case, the false negative rate no longer decreases once the model is guaranteed to see at least one frame of each event. In the empirical case, we perform 10,000 trials of sampling at each sample rate, and measure the false negative rates. By using the measured conditional probability of the dataset, Mainstream is able to produce the "Mainstream prediction" line that approximates the empirical line closely.

We use Mainstream for event detection but believe its approach can be generally applied to DNN-based tasks where application quality depends not only on its model, but also on its input sampling rate (e.g., object tracking, action recognition.)

Switching models vs. sharing computation. There are many DNN architectures representing different points in the throughput verses accuracy trade-off space that we discuss above. Interestingly, we find that Mainstream's approach of sharing computation among partially-specialized

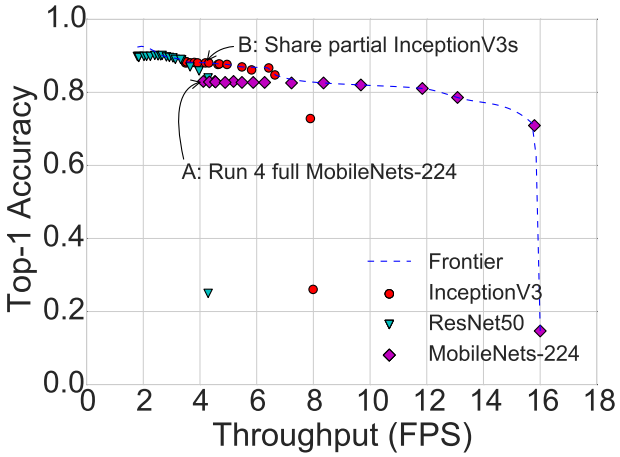


Figure 5.7: Accuracy vs. throughput for 3 DNN architectures.

DNNs is typically best. (Also, Mainstream's approach of finding the right configuration at application deployment time will be more robust than relying on the developer to select without knowing how much resource will be available based on competing applications.)

Figure 5.7 shows the Top-1 accuracy versus throughput when running inference for four concurrent applications. Each of the applications uses the same one of three popular DNN architectures. ResNet-50 [42] is the most accurate, and slowest; InceptionV3 [114] is somewhat less time-consuming; and MobileNets-224 [46] is the fastest. For each architecture, the left-most point is fully specialized (no sharing) and each successive point to the right shares more layers via Mainstream. Sharing more layers allows for higher system-wide throughput. The dotted Pareto efficient frontier shows potentially optimal combinations of accuracy and throughput.

Each of the architectures offers some Pareto efficient configurations, but the key take-away is that sometime the more expensive architecture can provide higher accuracy at the same throughput compared to a cheaper architecture with less sharing. For example, point A represents a traditional no sharing approach without Mainstream, where four independent applications are deployed on the edge node. That configuration achieves lower accuracy than point B, a sharing configuration found by Mainstream. The only Pareto efficient "no sharing" option is when maximum accuracy is desired at any cost to throughput (in which case, deploy four fully specialized ResNet-50s).

5.3 Mainstream architecture

We have developed Mainstream, a training and runtime system for DNN-based live analytics of camera streams, which (a) enables efficient sharing of computation between detection applications, (b) maximizes event F1-score across all tasks, and (c) allows each task to be independently

developed, trained, and deployed. Figure 5.1 shows the architecture of Mainstream, which consists of three components: M-Trainer, M-Scheduler, and M-Runner.

To deploy a new application to Mainstream, the user provides a corresponding labeled training dataset to their local instance of M-Trainer (Step 1). M-Trainer uses the dataset to train a number of potential models, with varying numbers of specialized layers. This *Model Set* and associated metadata are then assembled into an "M-Package" (Step 2). Note that these are offline steps, performed just once per application prior to deployment, independent of all other tasks. At runtime, M-Scheduler uses the M-Packages of all currently-deployed applications to determine, for each application, the degree of DNN sharing and sampling rate such that, across all applications, the event F1-score is maximized, subject to the resource limits of the edge platform (Step 3). M-Scheduler runs in the datacenter, and is executed once for each scheduling event (e.g., a change in the deployed set of applications, or in available hardware resources). M-Runner then executes the selected model configuration on edge devices (Step 4) and returns app-specific results in real-time (Step 5).

M-Runner is a relatively straightforward execution system for running visual processing pipelines. It accepts a DAG, where each node represents a unit of independent computation, and connections represent data flow. Figure 2.1 illustrates the logical DAG for an image classification application. Most of the computation is expected in the "DNN" process, which evaluates the merged DNN of all concurrent tasks. This combined DNN, as illustrated in Figure 5.3d, represents the set of models selected by M-Scheduler across all tasks. This DNN is structured as a tree, with sets of layers branching from the shared stem. M-Runner executes the shared stem once per frame, reducing the total processing costs of the deployed tasks.

We next describe how M-Trainer independently trains model candidates for potential sharing (Sec. 5.3.1) and how M-Scheduler dynamically chooses among them (Sec. 5.3.3).

5.3.1 Distributed sharing-aware training

M-Trainer produces a set of models for each task so that they can be combined dynamically at runtime to maximize collective performance. Application developers use M-Trainer independently at different times and locations.

One approach to sharing computation between applications would be to jointly train them using a multi-task network. This, however, requires centralized training of applications. MCDNN [40] proposed fine-tuning models independently and sharing the unspecialized DNN layers. This static approach prevents M-Scheduler from dynamically tailoring stem-sharing to the available resources. In contrast, M-Trainer generates a *set of models* that vary in the number of specialized layers. These models compose an application-specific *Model Set*. The generation of Model Sets allows for the late binding of the degree of specialization to deployment time, when the platform characteristics and co-deployed applications are known.

To construct a Model Set, M-Trainer first analyzes the structure of the base DNN to identify

branchpoints, the potential boundaries between frozen and specialized layers. Our heuristic marks branchpoints at layers which are connected to subsequent layers by a relatively small number of activations. Using this heuristic, the chokepoints correspond well to boundaries between blocks in modular network architectures such as ResNet and Inception, which are built from repeated modules of tightly coupled network layers.

Using the app-specific training data provided by the developer, M-Trainer generates a set of fine-tuned DNN models, one per branchpoint, where layers up to the branchpoint are frozen, and the rest are specialized. Only the models at the Pareto-optimal frontier with respect to number of layers specialized and estimated accuracy are actually included in the M-Package. This eliminates from consideration models that reduce accuracy, while requiring more specialization. For example, an overfitted model, caused by specializing too many layers with insufficient training data, will not be included.

Model Sets are bundled together with application metadata into an *M-Package*. This metadata includes the measured per-frame accuracy of each model (we use a portion of the data as the validation dataset.) The expected minimum event duration, event frequency, and inter-frame correlation are optionally measured from the training data and included in the M-Package, or directly provided by the application developer.

The construction of the M-Package is an offline operation, which is run just once per application. For each application, M-Trainer must train multiple models. Although training a model from scratch can be very resource intensive, fine-tuning is much quicker. M-Trainer creates Model Sets with 15 model options in 8 hours on a single GPU (Sect. 5.4). Note that the computation for generating all of the models is easily parallelized in a datacenter setting, and may not be significantly higher than traditional fine-tuning. For example, to find the right number of layers to specialize in order to maximize accuracy, one may need to generate these models anyway. The key difference here is that intermediate runs are not discarded, and the final selection is made at run time by M-Scheduler.

As each application's models are independently trained and analyzed, no coordination or sharing of training data is needed between developers of different tasks. The resulting M-Package, however, contains enough information that M-Scheduler, at run time, can optimize across the independently-developed tasks.

5.3.2 Hyperparameters and overfitting

During training, a user is required to tune the *hyperparameters* used (e.g. learning rate, decay). Hyperparameters are tunable knows chosen so as to prevent *overfitting* and ensure that the learned model converges. Training deep networks requires attention to hyperparameters and overfitting.

In practice, a single set of hyperparameters is effective across tasks when the same architecture is used. For the design of M-Trainer, we do a parameter sweep of different optimizers,

learning rates and decays. We find that the Adam optimizer, a learning rate of 0.0001 and a decay of 0.5 work well for M-Trainer using InceptionV3. To improve training, M-Trainer could be equipped to use dynamic hyperparameters or future improved optimizers. To reduce overfitting, our model includes batch normalization and dropout [93]. We use early-stopping so that we stop training once the validation loss stops decreasing.

5.3.3 Dynamic sharing-aware scheduling

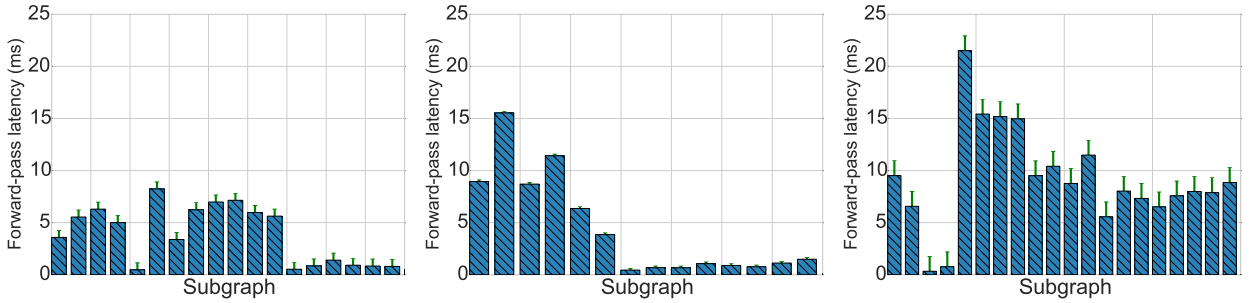
At each *scheduling event* (typically, an application submission or termination), M-Scheduler uses the M-Packages created by the various per-application M-Trainers to produce a new overall schedule that optimizes some global objective function, subject to resource constraints (currently, M-Scheduler maximizes average event F1-score across applications). The *schedule* consists of a DNN model selection (indicating the degree of sharing) and target frame-rate for every running application.[1] The final schedule is a tree-like model with applications splitting from a shared stem at potentially different branch points, with each application able to run at its own frame rate.

M-Scheduler optimization algorithm. With N applications to schedule, S possible specialization settings per application, and R frame-rate settings per application, the number of possible schedules is $(S \cdot R)^N$. Although this space is large (e.g., $N \approx 10$, $R \approx 10$, and $S \approx 10$ in our experiments), M-Scheduler can efficiently determine a good schedule using a greedy heuristic (Algo. 2). We compare the schedules generated by our greedy scheduler to those of an exhaustive scheduler in >4,800 workloads each consisting of up to 10 applications, and find that the greedy schedules are on average within 0.89% of optimal.

Algorithm 2 Scheduler optimization algorithm

```
function GET NEXT MOVE(schedule)
                                  ▷ Finds change to schedule with the highest benefit/cost
end function
function SCHEDULE(budget)
    sched ← GET SCHEDULE(max_sharing, min_fps)
    while True do
        next_move ← GET NEXT MOVE(sched)
        cost ← cost + GET COST(next_move)
        if cost < budget then
            sched ← UPDATE SCHEDULE(next_move)
        else return sched
        end if
    end while
end function
```

[1] Here, we assume that some admission control policy (outside the scope of this work) has been applied to ensure that some schedule is feasible for the set of running applications.



(a) InceptionV3 **(b)** MobileNets-224 **(c)** ResNet-50

Figure 5.8: Latency per layer per frame, used to calculate schedule cost.

Essentially, at each step of our iterative algorithm, the scheduler considers making a *move* which improves the application quality of a single application by tweaking its frame rate and/or model specialization. The algorithm greedily selects the move that yields the best ratio of *benefit* to *cost*, defined below. Naturally, before this iterative refinement, the schedule is initialized to the lowest cost configuration— Max-Sharing with minimum frame rate. At any iteration step, the number of possible moves is bounded by $S \cdot R \cdot N$. The total number of moves per invocation of the scheduler is similarly bounded by $S \cdot R \cdot N$, but in practice is much fewer as the set of potential moves that fit within the computational budget is exhausted.

Measuring the Benefit of a Move. The benefit associated with a move captures the improvement in F1-score for the application associated with that move. This value is calculated using the analytical model presented in Section 5.2 and the application metadata in the M-Package.

Measuring the Cost of a Move. The cost value considered represents the computational resources (e.g., CPU-seconds per second) consumed by a given schedule arrangement and depends on the number of shared subgraphs, the number of task-specific subgraphs, and the intended throughput (frame-rate) of each subgraph. The relative cost of a schedule is the sum of the execution time of each model layer, multiplied by the desired throughput. Consider for example a schedule with two applications, both executing at F FPS. Assume they share a compute stem A, and then branch to specialized subgraphs B_1 and B_2. If C_A represents the execution cost (in CPU-seconds per frame, say) of A, and C_B the execution cost of B_1 and B_2, then the total cost of the schedule is $F \cdot C_A + 2F \cdot C_B$. Adding a third application based on the same network, using the same branchpoint and frame rate will add another factor of $F \cdot C_B$ to the schedule cost.

To most accurately model the compute costs (e.g., C_A and C_B), a forward pass execution of the base DNN should be executed and measured once on the target hardware. Note that as cost is relatively insensitive to the assigned model weights, each base DNN need only run one time (ever) per target hardware, not once per trained application. Examples of the subgraph latency costs of InceptionV3, MobileNets-224 and ResNet-50 on an Intel NUC are shown in Figure 5.8.

Max-Min Fairness Among Applications. Although stem-sharing improves overall system efficiency, maximizing a global objective may lead to an inequitable allocation of resources for

individual applications. Thus, M-Scheduler can also be run in max-min fairness mode, which maximizes the minimum benefit among applications, instead of the average. Max-min fairness is scheduled by searching the space using dynamic programming.

Potential Threat Model We also briefly discuss the adversarial weaknesses of our resource allocator. Using the analytical model of M-Trainer, each application must supply the estimated accuracy, given how many of the specialized layers are used for inference. However, application developers could falsify parameters in order to gain an unfair share of the available resources. A potential future line of research is to monitor the discordent classifications of models with different levels of specialization, and use differences in model behavior to justify increased resource consumption.

X-Voting To Improve Precision. Mainstream uses *voting* across frames to improve precision, and consequentially F1-score. With X-voting, Mainstream requires X consecutive positives to identify an event. While X-voting decreases false positives, it is not guaranteed to increase precision. For X-voting to improve precision, it must decrease false positives at a higher rate than true positives. The ideal X-voting configuration again depends on the applications and the resources available. A higher X incurs fewer false positives, but requires more cost to sustain high recall (either by increasing FPS or increasing specialization). We evaluate the effect of various X-voting configurations in Sect. 5.5.

5.4 Experimental setup

To evaluate our system, we implement seven different event detection applications. We refer to the set of applications as 7-Hybrid. These are listed in Table 5.1, along with the datasets we used to train and test them. A pedestrian-detection application (Pedestrian) is trained based on the fully-labeled, publicly-available Urban Tracking video dataset [56]. Our application to classify car models (Car) uses the Stanford Cars image dataset [66]. Train detection (Train) is based on video of nearby train tracks that we have captured in our camera deployment, and have hand labeled. The remaining classifiers are trained on a video of a nearby intersection, also captured in our camera deployment. We have obtained the necessary permissions and plan to make our Trains and Intersection video dataset available publicly. We reserve a portion of these datasets to create synthetic video workloads for testing.

We use M-Trainer to produce a task-specific M-Package for each application. Model candidates are fine-tuned using the MobileNets-224 model pretrained on ImageNet as a base DNN (implemented in Keras [17] using TensorFlow [2]). Each M-Package contains several models with different degrees of fine-tuning as described in Section 5.3.1. We evaluate Mainstream using the M-Packages and hold-out validation sets from our datasets. Our experiments use the applications in Table 5.1.

Hardware. Training is performed on nodes equipped with Intel® Xeon® E5-2698Bv3 processors (2.0 GHz, 16 cores) and an Nvidia Titan X GPU. All end-to-end experiments use an Intel®

Detection Task	Dataset Description	Number of Images	Avg Event Length	Min Event Length	Event Frequency
Pedestrians	Urban Tracker atrium video	4538	59	2	0.63
Bus	Intersection near CMU video	4762	1039	141	0.27
Red Car	Intersection near CMU video	9172	228	46	0.08
Scramble	Intersection near CMU video	1500	412	382	0.16
Schoolbus	Intersection near CMU video	2600	854	92	0.03
Trains	Train tracks near CMU video	3066	132	20	0.01
Cars	Images of 23 car models	3042	—	—	—

Table 5.1: Labeled datasets used to train classifiers for event detection applications. Average and minimum event lengths are reported in number of frames. Event length and event frequency only apply to video datasets and not Cars.

NUC with an Intel® Core™ i7-6770HQ processor and 32 GiB DRAM, which is intended to represent an edge processing device. The Train and Intersection videos were captured using an Allied Vision Manta G-1236 GigE Vision camera.

5.5 Evaluation

We evaluate our system in the context of independent DNN-based video processing applications sharing a fixed-resource edge computer. In our evaluation, we show that Mainstream's dynamic approach outperforms static solutions in all of our experimental settings, across various application workloads, computational budgets, and numbers of concurrent applications. Mainstream's X-voting is capable of improving F1-score but, like model specialization, must also be dynamically configured to the resources available. In addition to our benchmarked applications, we show an end-to-end Mainstream deployment of a train detection application used for environmental pollution monitoring.

5.5.1 Mainstream improves video analysis application quality

Our goal in event detection is to maximize per-*event* F1-score. We compare the F1-score achieved by Mainstream with two baselines: No-Sharing and Max-Sharing. *No-Sharing* is the default behavior for a multi-tenant environment and is the approach taken by systems like TensorFlow Serving [1] and Clipper [21]. No-Sharing maximizes classification accuracy at the cost of a reduced sampling rate and requires no coordination between tenants. *Max-Sharing* is the sharing approach used by MCDNN [40]. It uses partial-DNN sharing by fine-tuning the final layer of concurrent DNNs. In many cases, Max-Sharing provides better F1-score relative to No-Sharing when a non-trivial number of applications share the infrastructure; it sacrifices classification accuracy to maximize the number of frames processed. We show, however, that Max-Sharing is less effective than making deliberate runtime decisions about how much sharing to use.

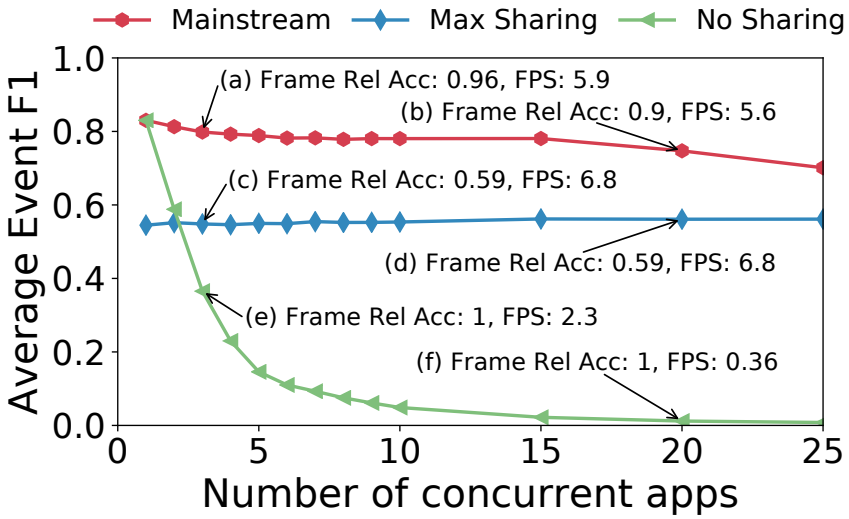
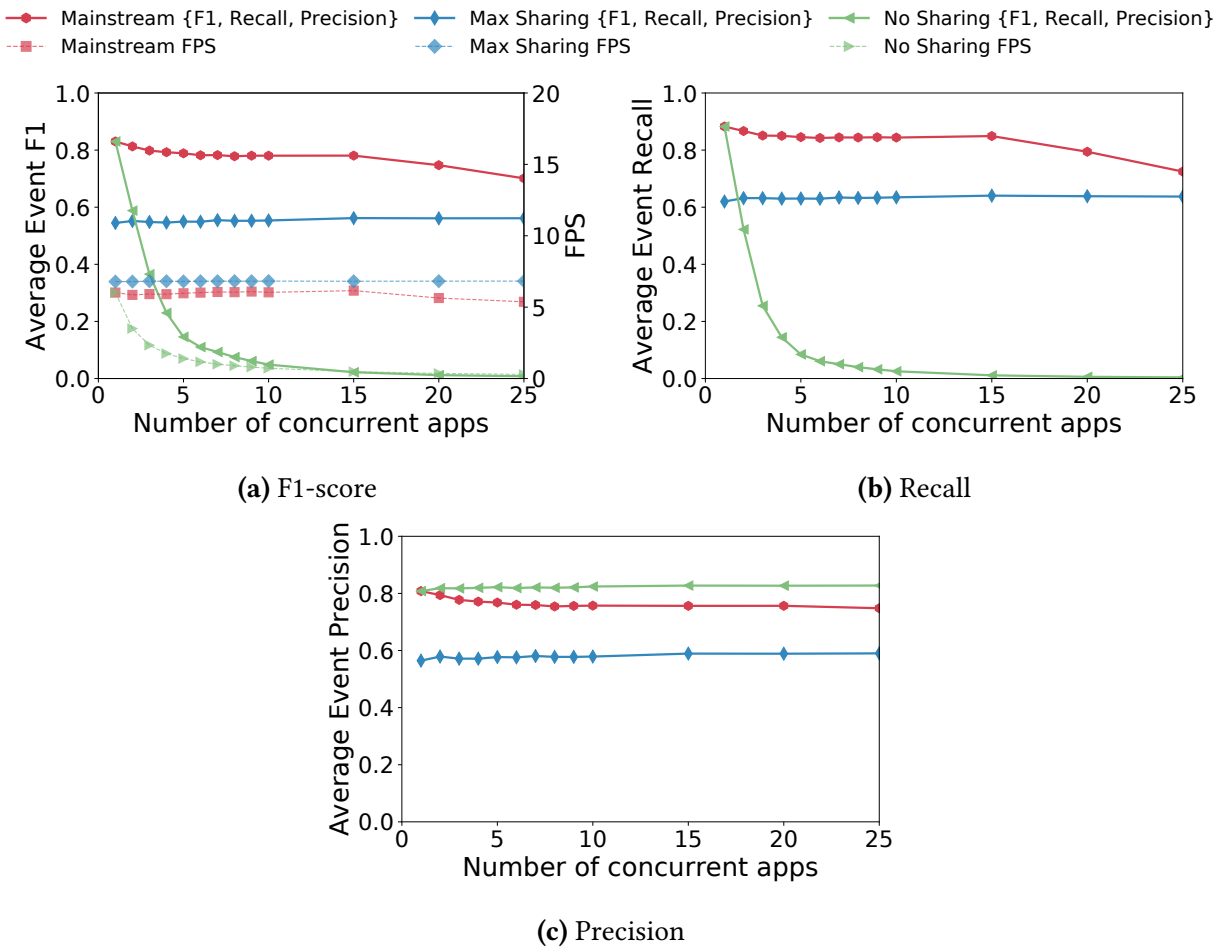


Figure 5.9: Mainstream improves F1-scores vs. No-Sharing between applications or conservatively sharing all layers but the last one. "Frame Rel Acc" is the relative image-level accuracy of the model deployed, compared to the best performing model candidate. "FPS" is the average observed throughput of the deployed applications.

In order to observe the effects of increasingly constrained resources without a large number of distinct applications, we generate additional applications by augmenting our application set. Each of the seven classification tasks in Table 5.1 has a corresponding "accuracy-tradeoff curve", which represents the relationship between per-frame accuracy and the shared stem size (Figure 5.4b). For each application in our experiments, we randomly choose one of the seven classifiers (and its corresponding accuracy-tradeoff curve) and parameterize it with a different event length, event frequency and inter-frame correlation. To capture the effects of diverse application characteristics, the parameters are uniformly sampled from a range of possible values. Each workload consists of up to 30 concurrent applications. In most experiments, we show the behavior averaged across 100 workloads. Our video capture rate for all experiments is 10 FPS.

Mainstream outperforms static approaches. M-Scheduler maximizes per-event F1-score by varying the sampling rate and amount of sharing. Each additional application introduces more resource contention, forcing the system to pick a different balance between accuracy and sampling rate to achieve the best average F1-score.

Figure 5.9 compares Mainstream with our baseline strategies. Mainstream delivers as much as a 87X higher per-event F1-score than No-Sharing and as much as a 47% higher score vs Max-Sharing. Figure 5.9 reports F1-scores averaged across 100 workloads. The relationship between the three schedulers holds when examining individual workloads. No-Sharing exhibits low recall because of its low throughput—the system has fewer opportunities to detect the event. Max-Sharing has high throughput but a worse precision because the underlying model accuracy is lower—it evaluates many frames but does so inaccurately. Mainstream outperforms by striking a balance, sometimes choosing a more accurate model, and sometimes choosing to run at a higher



(a) F1-score

(b) Recall

(c) Precision

Figure 5.10: The average event F1-score (Figure 5.10a), recall (Figure 5.10b) and precision (Figure 5.10c) across 100 deployed workloads are shown (solid lines) alongside the average frame-rate across applications (dotted lines). Mainstream dynamically balances recall and precision to maximize aggregate F1-score. With high numbers of concurrent applications, Mainstream sacrifices small amounts of both specialization and frame-rate.

throughput.

Mainstream dynamically balances precision and recall. Figure 5.10 delves into the system effects of Mainstream more deeply. F1-score, recall, and precision are plotted. The average application frame rate is plotted, showing how Mainstream dynamically tailors resource usage to the workload. (Not shown is the varying model accuracy.) Optimizing for precision requires careful tuning of the application frame rate. While higher FPS always leads to higher recall, it does not always lead to higher precision. (A high frame rate may only increase false positives without increasing expected true positives.) For instance, No-Sharing's low frame rate and high per-frame accuracy allows it to have the highest precision of the three approaches. When given just a few applications, Mainstream runs specialized models, while throttling the stream rate to avoid unnecessary false positives. As resources become scarce, many applications begin to share more of the network.

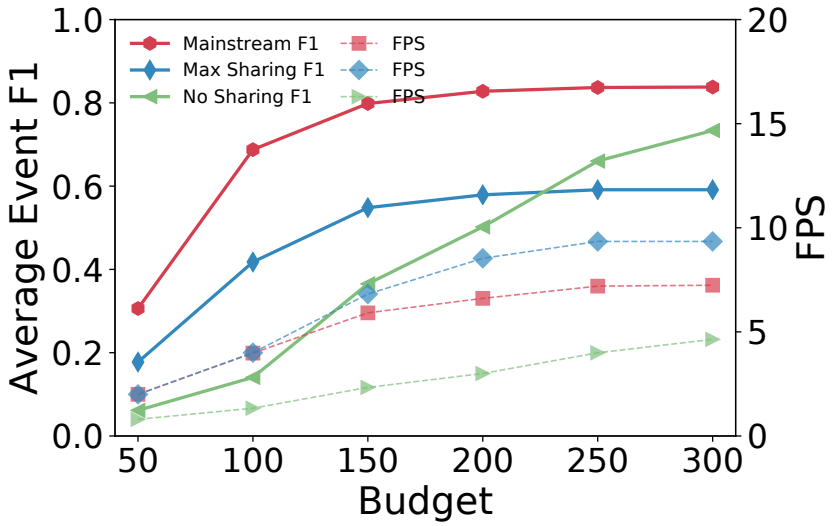
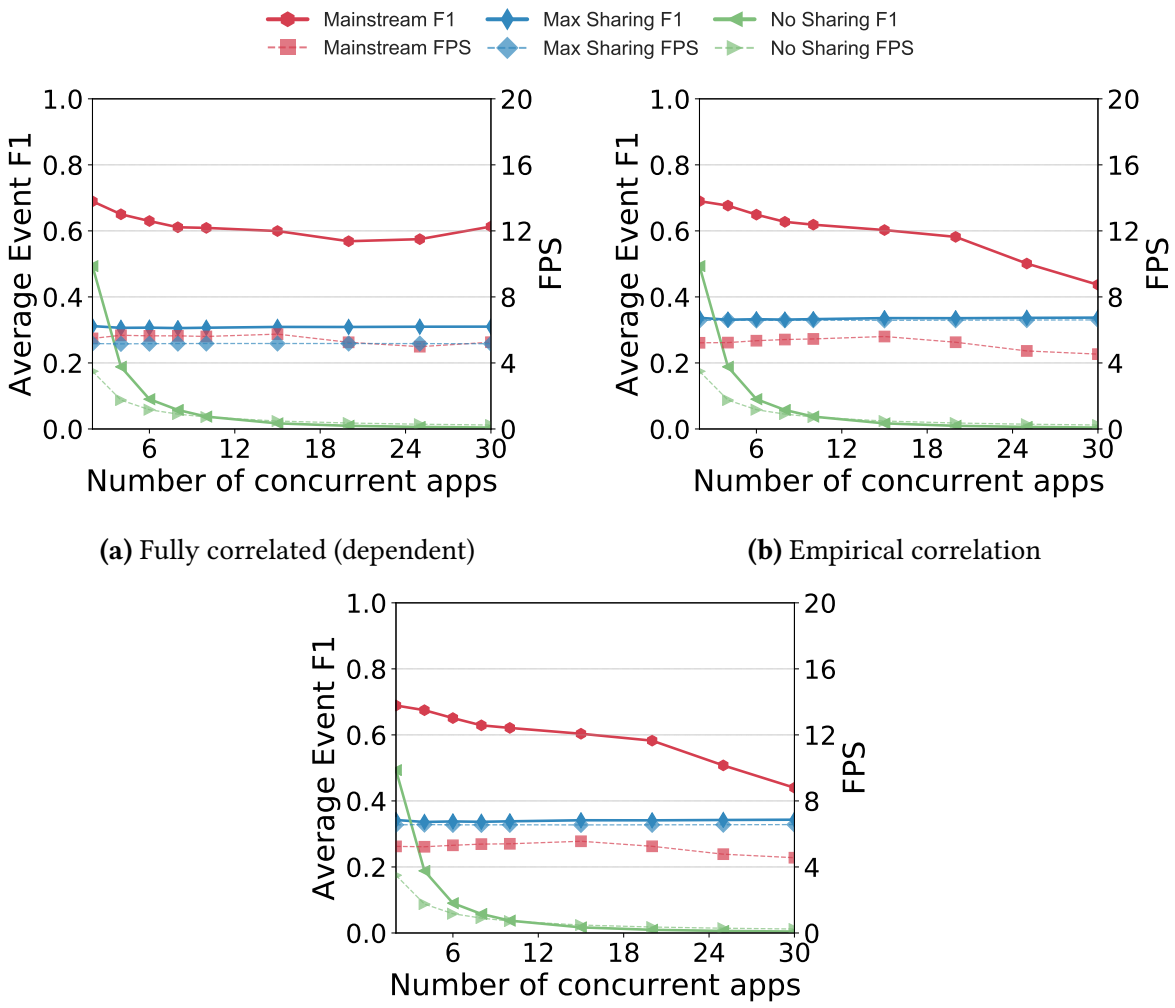


Figure 5.11: Mainstream improves F1-score of workloads under varying computational budgets. The rightmost points on the X axis represent the resources available on an Intel® NUC. Even under heavy resource constraints, there is available capacity for Mainstream to perform optimizations.

Mainstream improves upon Max-Sharing even under tight resource constraints. Figure 5.11 shows the effect of Mainstream, Max-Sharing, and No-Sharing on a range of computational budgets. We average the event F1-scores across 100 workloads, each with 3 applications. The right-most points represent the scenario of running on computational resources equivalent to an Intel® NUC. With a small workload of three applications, No-Sharing performs better than Max-Sharing, as it is able to run expensive models at a high enough frame rate. As we decrease the available budget, Max-Sharing's conservative sharing approach allows it to be more scalable than No-Sharing. However, even after the computational budget is reduced by 83%, Mainstream still improves application performance, compared to the overly conservative Max-Sharing approach.

5.5.2 Robustness to inter-frame correlation

Mainstream uses inter-frame correlation to model the benefit of increasing an application's frame-rate to its event F1-score. We show the range of behavior affected by the inter-frame correlation in Figure 5.12. With fully uncorrelated event frames (Figure 5.12c), the probability of classifying a frame is independent of the result of previously evaluated frames. With fully correlated event frames (Figure 5.12a), applications get no benefit from sampling an event multiple times. In practice, tasks will lie somewhere in between. In Figure 5.12b, we use empirical correlation values derived from the videos in 7-Hybrid. When frames are more uncorrelated, increasing frame rate is more effective for improving recall than increasing accuracy. Both Mainstream and Max-Sharing take advantage of uncorrelated frames by increasing application frame rate. However, when frames within an event are uncorrelated, both true and false positives are easier to come



(a) Fully correlated (dependent)

(b) Empirical correlation

(c) Fully uncorrelated (independent)

Figure 5.12: The inter-frame correlation significantly affects application quality. We run Mainstream, No-Sharing and Max-Sharing with the two extremes: fully uncorrelated event frames (Figure 5.12a), and fully correlated event frames (Figure 5.12c). In practice, the inter-frame correlation is somewhere in between (Figure 5.12b). Mainstream provides improved F1-score across the spectrum of correlations values.

by. Max-Sharing, which is running an inaccurate model with a high false positive rate, runs at a lower frame rate in the fully correlated case, compared to the empirical case to increase its precision. For the same reason, in the fully correlated case, Max-Sharing runs at a lower frame rate than Mainstream. The varying schedules between the three experimental configurations show the importance of modeling inter-frame correlation. Under a variety of inter-frame correlation, Mainstream is again able to find F1-score-improving conditions.

5.5.3 Tuned X-voting improves F1-score

Applications that suffer from low per-frame precision will generate many false positives. An X-voting approach can greatly decrease the incidence of false positives, as X consecutive classifications are needed in order to report a detection. Too large a value of X can hurt recall, causing real events to go unreported. By using X-voting and optimizing the parameter X, Mainstream can improve the overall average event F1-score.

Figure 5.13 shows the effects of X-voting on F1-scores as X and the number of applications are varied, while total resources are kept fixed. With just a few concurrent applications, running at high frame rates, 7-voting and 5-voting yield the highest F1-scores. With more resource contention, and lower throughput, 3-voting becomes the best choice as the cost of dropping true positives outweighs the benefit of reducing false positives for higher values of X. When resources become too constrained, this approach is less viable, e.g., 1-voting becomes the best approach at 25 concurrent apps. Figure 5.13 also shows the Pareto frontier of F1-scores achievable across all values of X for a given number of concurrent applications.

5.5.4 Mainstream deployment

We deployed our environmental pollution monitor application and nine other concurrent applications using both Mainstream and a conventional No-Sharing approach for one week on the hardware setup described in Chapter 5.4. Figure 5.14 shows the trace of both approaches on a *single* train event sequence, indicating the frames analyzed. A hit represents a correct classification of the train, a miss represents an incorrect classification. We see that Mainstream's deployment samples the stream more frequently, yielding many more hits (and misses) than No-Sharing; the result, though, is that Mainstream detects the train event earlier and more confidently.

We control the false positive rate with 2-voting, requiring Mainstream to have two positive samples before an event is classified. The false positive rate of the Train video drops from 0.028 to 0.00056. No-Sharing and Mainstream achieve a 0 and a 0.00056 false positive rate, respectively. In the analyzed deployment in Figure 5.14, we see that Mainstream still detects the train easily and quickly.

False Positive Frequency. When evaluating the effectiveness of an inference system, one must consider how the system reports false positives in addition to its false negative reporting.

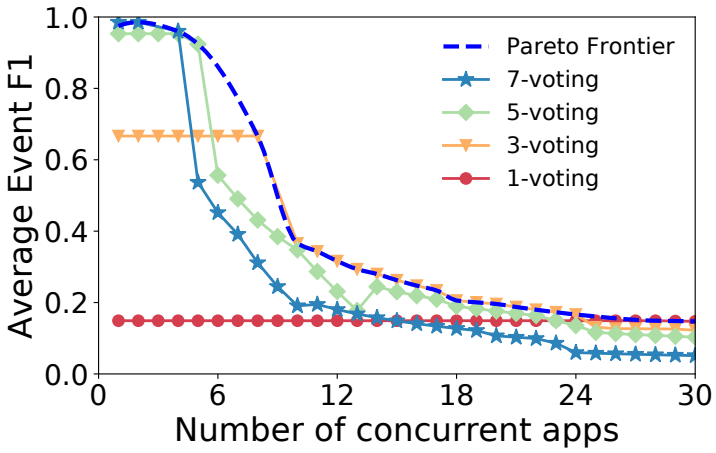


Figure 5.13: X-voting increases precision and helps Mainstream achieve a higher F1-score, but only if frame rate is high enough to avoid hurting recall. Thus, the effects vary by the level of resource contention. The Pareto frontier shows the F1-scores achievable given the dynamic selection of an optimal X-voting scheme for the resource scenario.

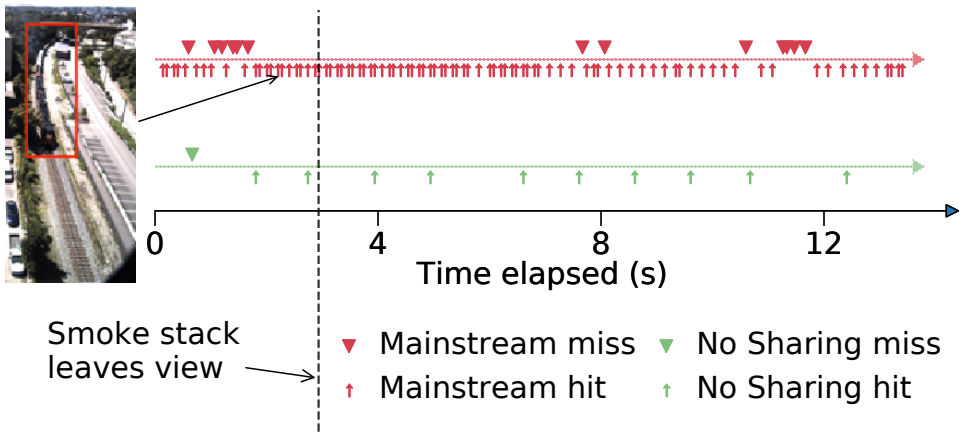


Figure 5.14: Timeline of Mainstream running a train detector app with 9 concurrent applications. Our goal is to detect the train as early as possible, before the smoke stack is out of view (end of window represented by the dotted line). Mainstream detects the train earlier and more confidently than No-Sharing.

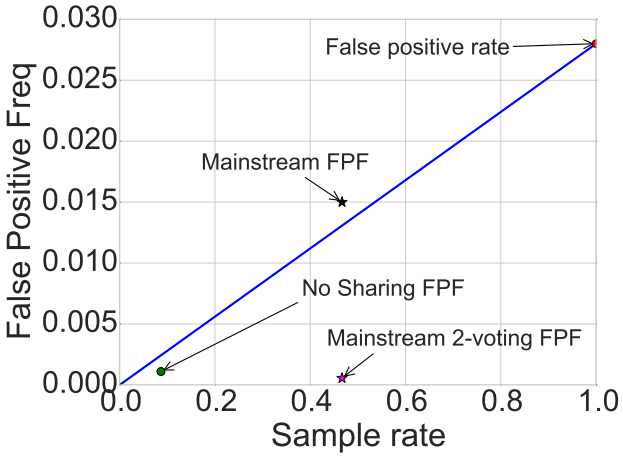


Figure 5.15: False positive frequency is a function of the per-frame false positive rate and the sample rate. The FP frequency increases as more frames are sampled. The FP frequency at sample rate of 1 is equal to the per-frame false positive rate. The data is from the Train video. Also included is the false positive frequency of No-Sharing and Mainstream.

However, reporting false positives over time is not completely analogous to FNR. For an event detection system, the FNR is naturally captured as the number of detected events divided by the number of actual events.

In contrast, the most natural way to consider false positives is to measure the number of false alarms reported. That is, how often does the system report the presence of an object (e.g., train) when one is actually not visible in-frame. We term this measure the *false positive frequency*, and it is affected both by the per-frame false positive rate and the frame sampling rate. Both can be affected by the choices that Mainstream makes.

Figure 5.15 shows how the sampling rate affects the false positive frequency in our application environment. The rates shown are from frames of the Train video which do not include a train. The per-frame false positive rate of the deployed model is 0.028. This is equal to the false positive frequency at a sampling rate of 1. Neither No-Sharing nor Mainstream has as high of a false positive frequency, because their sampling rates are less than 1 from necessity. Since No-Sharing runs at a much lower frame rate than Mainstream, its false positive frequency is correspondingly lower.

To control the false positive rate, we can use the "X-voting" discussed in Sect. 5.3.3. "X-Voting" requires Mainstream to have multiple positive samples before an event is classified. Even with just 2-voting, the false positive rate of the Train video drops from 0.028 to 0.00056. No-Sharing and Mainstream achieve a 0 and a 0.00056 false positive frequency, respectively. In the analyzed deployment in Figure 5.14, we see that Mainstream has no trouble still detecting the train, while No-Sharing does not detect 2-consecutive positive samples in the presence of the train.

5.6 Additional related work

Several recent systems have attempted to tackle the problem of optimizing execution of visual computing pipelines.

VideoStorm [125] is a video analytics system for large-scale clusters and workloads. It analyzes resource use and application-goal-based metrics as a function of tunable parameters of the analytics pipelines, building models for each application independently. It uses these models to allocate resources and select parameters for deployed applications on a target platform, in order to maximize application quality metrics. VideoStorm takes a black-box view of the applications, and assumes that quality and resource consumption of co-deployed applications are independently determined. Therefore, it cannot take into account computation sharing, or optimize the sharing vs. degree of specialization tradeoff. In contrast, Mainstream takes a white-box approach to modeling application quality, and can explicitly tune computation sharing to improve application quality metrics. Compared to VideoStorm, Mainstream sacrifices some generality to solve the joint optimization problem.

MCDNN [40] introduces a static approach to sharing DNN computation, in which each application developer independently determines their amount of model specialization. MCDNN opportunistically shares any identical unspecialized layers between applications. In contrast, Mainstream's training and scheduling components allow late binding and jointly-optimized selection of the degrees of specialization at run time, when resource availability and co-deployed tasks are known.

Inference serving systems. Mainstream is an inference serving system for running neural networks on resource-constrained nodes. Other inference serving systems include Clipper [21], NoScope [59], and TensorFlow Serving [1]. Like Mainstream, these systems optimize for latency and throughput gains. Clipper caches results from multiple models, dynamically chooses from the results, and optimizes the batch size. NoScope replaces expensive neural networks for object detection with cheaper difference detectors and specialized models. TensorFlow Serving increases throughput with batching and hardware acceleration. LASER [4] and Velox [20] are inference serving systems for non-DNN models. LASER deploys linear models while Velox deploys personalized prediction algorithms using Apache Spark.

Unlike Mainstream, these inference serving systems do not share computation between independently trained models. They also target cluster environments. Mainstream targets edge devices with limited resources, where achieving the right degree of DNN computation sharing is particularly important, though such sharing would also be valuable in large data centers.

Reducing DNN inference time. Approaches to reducing DNN inference time for vision applications can be broadly classified into those that reduce model precision [15, 18, 19, 36, 51, 97, 128], use efficient network architectures [46, 53], use anytime prediction methods [47, 49], or employ model compression and sparsification [79, 119]. All of these methods are orthogonal to Mainstream's adaptive DNN computation sharing technique, but share its goal of selecting the

right trade-off between per-frame quality and frame throughput.

Multi-task networks. Multi-task learning [5, 12, 65, 84, 88, 91, 96, 127] is a ML approach in which a single model is trained to perform multiple tasks. Using multiple tasks to train a single model helps achieve better accuracy because of better generalization and complementary information [12, 103]. In the context of DNNs, a multi-task network can have a varying number of shared layers across tasks and task-specific layers [88, 91]. Multi-task learning assumes that all of the tasks are known *a priori*, and that training data for all of the tasks is available for use in a single training process. In contrast, Mainstream allows each task to be developed, trained, and deployed independently, and avoids the need to share or expose proprietary or privacy-sensitive training data between task developers. Note that one can run a multi-task network as a single large application in Mainstream.

Chapter 6

Conclusion and Future Directions

We conclude this thesis with a summary of the contributions and results from our research, limitations of our work, and potential future research directions.

6.1 Contributions and research results

In this thesis, we provid supporting evidence that dynamic optimization of hyperparameters can improve deep learning training and inference. While dynamic hyperparameter optimization increases efficiency in a part of the training and inference pipeline, the optimization itself introduces a new cost. A major challenge in designing our dynamic optimizations is choosing the right metric to optimize for that provides an overall net benefit of efficiency. We show that by doing so, we can accelerate training by up to 3.5x and improve video application quality by up to 87x compared to traditional static approaches.

6.1.1 Selective-Backprop: Dynamically selecting training examples

Selective-Backprop accelerates training of neural networks by skipping the backwards pass for examples where the forward pass loss function indicates little value at a particular stage of training.

Increased sample efficiency. Our exploration of the effects of prioritizing challenging examples during training in Selective-Backprop shows that an example's training loss serves as a useful proxy to measure how challenging the example is for the network to classify correctly. The relative training loss of an example fluctuates over the course of training, especially when those examples are held out for a subset of epochs. Experiments on several datasets and networks show that Selective-Backprop finds examples that are more sample-efficient and converges to target error rates up to 3.5x faster than with standard SGD and 1.02–1.8x faster than the state-of-the-art sampling approach.

Example selection cost. Selective-Backprop's selection pass uses the latest model parameters to compute up-to-date losses for all training examples considered. This allows Selective-Backprop to dynamically adapt to changes in example efficiency. However, this approach requires additional selection forward passes not used directly for training. We define and evaluate a Selective-Backprop variant called StaleSB that executes selection forward passes every n^{th} epoch. This as an example where we needed to compromise between on dynamism and increased overhead, to maximize net efficiency. Compared to Selective-Backprop, selecting examples with stale loss information further accelerates training on average by 26%.

6.1.2 AdaptSB: Dynamically adapting example prioritization

AdaptSB generalizes Selective-Backprop by allowing tuning of the prioritization function, a mapping between the relative loss of the example to the likelihood of its backprop.

Improving sample efficiency. We use AdaptSB to explore the effects of tuning the prioritization of challenging examples during training. We find that the optimal prioritization function depends on the training dataset as well as the training time. Specifically, AdaptSB outperforms Selective-Backprop on datasets with label error by de-emphasizing the highest loss examples. On CIFAR10, AdaptSB outperforms Selective-Backprop by increasingly prioritizing challenging examples as training continues and the network is able to classify easy examples better.

AdaptSB introduces hyperparameter search cost. AdaptSB determines a priority function that minimizes wall-clock time by choosing examples with high sample-efficiency. Currently, AdaptSB chooses a priority function using a hyperparameter search. We find that this leads to improved wall-clock time once the hyperparameter search is completed. However, an end-to-end evaluation requires taking into account the overhead of the search itself. In our evaluation, we show that a potential solution is to perform the hyperparameter search on a subset of the training dataset, to reduce the relative cost compared to training. We also outline future approaches to reduce the absolute cost of performing the search, including techniques to reduce the size of the search space.

6.1.3 Mainstream: Dynamically tuning DNN specialization

Mainstream adaptively orchestrates DNN stem-sharing among concurrent video processing applications sharing the limited resources on an edge device.

Mainstream's model training and scheduling overheads. The goal of Mainstream is to maximize aggregate application quality—termed the event F1-score—across deployed applications. It does so by optimizing the degree of specialization for each application at runtime. Unlike traditional training, M-Trainer trains many models for a given task, and keeps "less optimal" candidate DNN models. This increases training time cost by having to train multiple (≈10) models. The M-Scheduler determines the best candidate for each application, including the degrees of

specialization and, thus, sharing. This require searching the option space to maximize application quality (e.g., average F1-score among the applications). We exclude the M-Trainer and M-Scheduler cost in the end-to-end metric because they are one-time costs and can be performed in the datacenter. By contrast, Mainstream applications run on a resource-constrained edge device and are typically deployed for several months. The one-time cost of training and scheduling is therefore quickly amortized. Overall, Maintream's overheads are much smaller compared to Selective-Backprop and AdaptSB.

Improving aggregate application quality. Mainstream provides scalability to edge deployments by sharing a portion of the DNN computation between applications. The key challenge to coordinating this resource sharing is to determine how much each application should specialize, and thus not share, their DNN. We find that the optimal configuration depends on runtime characteristics, namely the resources available to the concurrently deployed applications. We defer the determination to runtime by training multiple models with different amounts of sharing potential for each task. Mainstream is then able to choose which model to deploy. Its dynamic optimization results in much higher aggregate application quality, compared to static approaches. Experiments with several event detection tasks confirm that Mainstream significantly increases overall event F1-score relative to current approaches over a range of concurrency levels.

6.2 Limitations and future directions

We conclude by discussing possible future directions to expand the potential realized from the approaches developed in the three proposed systems.

Selective-Backprop for continuous learning. Catastrophic forgetting is a common challenge of applying continuous or incremental learning to high-dimensional data streams [48, 62, 83]. DNNs that are capable of incrementally performing new tasks or assimilating additional data are more efficient than models that must be retrained from scratch each time new information is available. However, DNNs are prone to performance degradation on their original task, once a new task is introduced.

One potential solution is to re-introduce data from old datasets when training on a new task but this can become prohibitively expensive. Incremental learning approaches must balance the use of new and old data to learn efficiently, yet still be able to characterize examples from old datasets. Selective-Backprop and AdaptSB dynamically identify examples useful at a given stage in training. One potential research direction is to use Selective-Backprop to improve continuous learning by dynamically sampling useful examples for training. Over the course of training, Selective-Backprop can reintroduce examples that the network has forgotten, but skip examples it still remembers. AdaptSB could also be used to tune the emphasis on old and new datapoints, which is likely to change between training tasks.

Distinguishing between out-of-distribution and mislabeled examples with AdaptSB. Examples exhibit a high loss for different reasons. Our work on Selective-Backprop and AdaptSB

shows that primarily two types of examples exhibit high loss: (1) examples that are under-represented and useful, and (2) mislabeled examples that are harmful to training. An important weakness of AdaptSB is that in a dataset contained mislabeled examples, AdaptSB de-emphasizes all high-loss examples, potentially skipping even useful under-represented examples that are necessary to improve generalization. An interesting future line of research would be to create better sampling methods that directly target useful examples by distinguishing between different types of challenging examples. Such a technique would likely also be useful in accurately flagging potential mislabels during dataset sanitation.

An analysis of intermediate activations is a potential approach to distinguishing under represented examples from mislabeled examples. The key idea is as follows: Unlike under-represented examples, mislabels are likely to exhibit similar activations as other examples in the dataset. By clustering similar intermediate activations, one could find under-represented examples by finding high-loss examples that are far from a cluster's centroid. Previously proposed approaches for detecting out-of-distribution examples are also promising. These include using a static threshold on maximum softmax confidence [44], training a network with either synthetic out-of-distribution examples [37, 72], or using an alternative dataset to detect outliers [44].

Mainstream beyond event detection. In this thesis, we used Mainstream for one application: event detection in video streams. A promising direction for future work is applying Mainstream's approach to a broader class of applications. Generally, Mainstream may be beneficial to other deep learning applications whose performance depends on both accuracy and throughput. For example, models performing object tracking must consider both accuracy and speed to track an object more closely across frames [80, 118, 120]. In order to use Mainstream in another domain, one must create an analytical model that determines the marginal benefit of improving accuracy or throughput to the final application's performance.

Bibliography

[1] Tensorflow Serving. https://www.tensorflow.org/serving/.

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

[3] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. YouTube-8M: A large-scale video classification benchmark. *CoRR*, abs/1609.08675, 2016.

[4] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14. ACM, 2014.

[5] K. Ahmed and L. Torresani. Branchconnect: Large-scale visual recognition with learned branch connections. *CoRR*, abs/1704.06010, 2017.

[6] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint*, arXiv:1511.06481, Nov 2015.

[7] S. Azadi, J. Feng, S. Jegelka, and T. Darrell. Auxiliary image regularization for deep cnns with noisy labels. 11 2015.

[8] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 02 2018.

[9] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning (ICML)*, 2009.

[10] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signSGD: Compressed Optimisation for Non-Convex Problems. In *International Conference on Machine Learning (ICML)*, 2018.

[11] D. Britz, A. Goldie, M.-T. Luong, and Q. Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

[12] R. Caruna. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[13] H.-S. Chang, E. G. Learned-Miller, and A. McCallum. Active Bias: Training a more accurate neural network by emphasizing high variance samples. In *Advances in Neural Information Processing Systems*, 2017.

[14] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 2002.

[15] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15. JMLR.org, 2015.

[16] S. Chintala. Convnet-Benchmarks. https://github.com/soumith/convnet-benchmarks.

[17] F. Chollet et al. Keras. https://keras.io, 2015.

[18] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *Advances in Neural Information Processing Systems*, 2016.

[19] M. Courbariaux, Y. Bengio, and J. David. Binaryconnect: Training deep neural networks with binary weights during propagations. 2015.

[20] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. In *CIDR*, 2015.

[21] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017. USENIX Association.

[22] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[24] T. Dettmers. TPUs vs GPUs for Transformers (BERT). https://timdettmers.com/2018/10/17/tpus-vs-gpus-for-transformers-bert/.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] T. DeVries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[27] S. Dodge and L. Karam. Understanding how image quality affects deep neural networks. In *Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2016.

[28] S. Doe, D. Nguyen, C. Stawarz, B. Refsdal, A. Siemiginowska, D. Burke, I. Evans, J. Evans, J. McDowell, J. Houck, and M. Nowak. Developing Sherpa with Python. In R. A. Shaw, F. Hill, and D. J. Bell, editors, *Astronomical Data Analysis Software and Systems XVI*, Astro-

nomical Society of the Pacific Conference Series, 2007.

[29] M. Ducoffe and F. Precioso. Adversarial active learning for deep networks: a margin based approach. *CoRR*, abs/1802.09841, 2018.

[30] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pages 2366–2374, 2014.

[31] Y. Gal, R. Islam, and Z. Ghahramani. Deep bayesian active learning with image data. In *International Conference on Machine Learning (ICML)*, 2017.

[32] J. Gao, H. V. Jagadish, and B. C. Ooi. Active Sampler: Light-weight accelerator for complex data analytics at scale. *arXiv preprint*, arXiv:1512.03880, 2015.

[33] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 2(4):22–33, 2003.

[34] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 38(1):142–158, 2016.

[35] J. Goldberger and E. Ben-Reuven. Training deep neural-networks using a noise adaptation layer. In *ICLR*, 2017.

[36] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014.

[37] D. Hafner, D. Tran, T. Lillicrap, A. Irpan, and J. Davidson. Reliable uncertainty estimates in deep neural networks using noise contrastive priors, 2019.

[38] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16. IEEE, 2016.

[39] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR 2016*.

[40] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Conference MobiSys'16 The 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobieSys '16. ACM, 2016.

[41] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034, 2015.

[42] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[43] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[44] D. Hendrycks and K. Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *Proceedings of International Conference on Learning Representations*, 2017.

[45] G. E. Hinton. To recognize shapes, first learn to generate images. In *Computational Neuroscience: Theoretical Insights into Brain Function*, Progress in Brain Research. Elsevier, 2007.

[46] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[47] H. Hu, D. Dey, J. A. Bagnell, and M. Hebert. Anytime neural networks via joint optimization of auxiliary losses. *arXiv preprint arXiv:1708.06832*, 2017.

[48] W. Hu, Z. Lin, B. Liu, C. Tao, Z. Tao, J. Ma, D. Zhao, and R. Yan. Overcoming catastrophic forgetting for continual learning via model adaptation. In *ICLR*, 2019.

[49] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *arXiv preprint arXiv:1703.09844*, 2017.

[50] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Computer Vision and Pattern Recognition (CVPR)*, 2017.

[51] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, 2016.

[52] J. Hui. Object Detection Series. https://medium.com/@jonathan_hui/object-detection-series-24d03a12f904.

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

[54] L. Jiang, D. Meng, Q. Zhao, S. Shan, and A. G. Hauptmann. Self-paced curriculum learning. In *Association for the Adancement of Artificial Intelligence (AAAI)*, 2015.

[55] L. Jiang, Z. Zhou, T. Leung, L.-J. Li, and L. Fei-Fei. MentorNet: Learning data-driven curriculum for very deep neural networks on corrupted labels. In *International Conference on Machine Learning (ICML)*, 2018.

[56] Jodoin, J.-P., Bilodeau, G.-A., and N. Saunier. Urban tracker: Multiple object tracking in urban mixed traffic. In *IEEE Winter conference on Applications of Computer Vision (WACV14)*, March 2014.

[57] T. B. Johnson and C. Guestrin. Training deep models faster with robust, approximate importance sampling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[58] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross,

M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[59] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. In *Proceedings of the VLDB Endowment, Vol. 10, No. 11*, 2017.

[60] A. Katharopoulos and F. Fleuret. Biased importance sampling for deep neural network training. *arXiv preprint arXiv:1706.00043*, 2017.

[61] A. Katharopoulos and F. Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *International Conference on Machine Learning (ICML)*, 2018.

[62] R. Kemker, A. Abitino, M. McClure, and C. Kanan. Measuring catastrophic forgetting in neural networks. *ArXiv*, abs/1708.02072, 2018.

[63] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[64] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[65] I. Kokkinos. Ubernet: Training a 'universal' convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. *CoRR*, abs/1609.02132, 2016.

[66] J. Krause, M. Stark, J. Deng, and L. Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[67] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

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

[69] M. Kuchnik and V. Smith. Efficient augmentation via data subsampling. In *International Conference on Learning Representations*, 2019.

[70] M. P. Kumar, B. Packer, and D. Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2010.

[71] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[72] K. Lee, H. Lee, K. Lee, and J. Shin. Training confidence-calibrated classifiers for detecting out-of-distribution samples. In *International Conference on Learning Representations*, 2018.

[73] T. J. Lee, J. Gottschlich, N. Tatbul, E. Metcalf, and S. Zdonik. Precision and recall for range-

based anomaly detection. *SysML*, Feb 2018.

[74] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 2017.

[75] Y. Li, J. Yang, Y. Song, L. Cao, J. Luo, and J. Li. Learning from noisy labels with distillation. 2017.

[76] M. Liao. Benchmarking Hardware for CNN Inference in 2018. https://towardsdatascience.com/benchmarking_hardware_for_cnn_inference_in_2018_1d58268de12a.

[77] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. pages 740–755", 2014.

[78] T.-Y. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *International Conference on Computer Vision (ICCV)*, 2017.

[79] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy. Sparse convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[80] Q. Liu, B. Liu, Y. Wu, W. Li, and N. Yu. Real-time online multi-object tracking in compressed domain. *IEEE Access*, 2019.

[81] T. Liu and D. Tao. Classification with noisy labels by importance reweighting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[82] G. LLC. Edge TPU performance benchmarks. https://coral.ai/docs/edgetpu/benchmarks.

[83] V. Lomonaco and D. Maltoni. Core50: a new dataset and benchmark for continuous object recognition. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, Proceedings of Machine Learning Research. PMLR, 2017.

[84] M. Long and J. Wang. Learning multiple tasks with deep relationship networks. *CoRR*, abs/1506.02117, 2015.

[85] I. Loshchilov and F. Hutter. Online batch selection for faster training of neural networks. *International Conference on Learning Representations (ICLR)*, 2015.

[86] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016.

[87] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[88] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. S. Feris. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *CVPR*, 2016.

[89] F. Ma, D. Meng, Q. Xie, Z. Li, and X. Dong. Self-paced co-training. In *International Conference on Machine Learning (ICML)*, 2017.

[90] T. Malisiewicz, A. Gupta, and A. A. Efros. Ensemble of exemplar-svms for object detection

and beyond. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11. IEEE Computer Society, 2011.

[91] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert. Cross-stitch Networks for Multi-task Learning. In *CVPR*, 2016.

[92] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[93] N.Srivastava, G. Hinton, A. Krizhevsky, I.Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. In *JLMR*, 2014.

[94] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[95] N. Papernot and P. McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[96] R. Ranjan, V. M. Patel, and R. Chellappa. Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. *arXiv preprint arXiv:1603.01249*, 2016.

[97] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[98] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: An astounding baseline for recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2014, Columbus, OH, USA, June 23-28, 2014*, pages 512–519, 2014.

[99] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018.

[100] S. E. Reed, H. Lee, D. Anguelov, C. Szegedy, D. Erhan, and A. Rabinovich. Training deep neural networks on noisy labels with bootstrapping. In *ICLR 2015*, 2015.

[101] M. Ren, W. Zeng, B. Yang, and R. Urtasun. Learning to reweight examples for robust deep learning. In *International Conference on Machine Learning (ICML)*, 2018.

[102] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Comput.*, 2004.

[103] S. Ruder. An overview of multi-task learning in deep neural networks. abs/1706.05098, 2017.

[104] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[105] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

[106] O. Sener and S. Savarese. Active learning for convolutional neural networks: A core-set

approach. In *International Conference on Learning Representations (ICLR)*, 2018.

[107] S. Seong, Y. Lee, Y. Kee, D. Han, and J. Kim. Towards flatter loss surface via nonmonotonic learning rate scheduling. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, pages 1020–1030, 2018.

[108] Y. Shen, H. Yun, Z. C. Lipton, Y. Kronrod, and A. Anandkumar. Deep active learning for named entity recognition. In *International Conference on Learning Representations (ICLR)*, 2018.

[109] A. Shrivastava, A. Gupta, and R. Girshick. Training region-based object detectors with online hard example mining. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[110] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*, pages 568–576, 2014.

[111] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[112] L. N. Smith. Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, 2017.

[113] L. N. Smith and N. Topin. Super-convergence: Very fast training of residual networks using large learning rates, 2018.

[114] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[115] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *CVPR*, pages 1701–1708, Washington, DC, USA, 2014. IEEE Computer Society.

[116] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. Technical report, 2012.

[117] K. Wang, D. Zhang, Y. Li, R. Zhang, and L. Lin. Cost-effective active learning for deep image classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 2016.

[118] Q. Wang, L. Zhang, L. Bertinetto, W. Hu, and P. H. Torr. Fast online object tracking and segmentation: A unifying approach. *arXiv preprint arXiv:1812.05050*, 2018.

[119] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. *Advances in Neural Information Processing Systems*, 2016.

[120] Y. Wu, J. Lim, and M. Yang. Online object tracking: A benchmark. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[121] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine trans-

lation system: Bridging the gap between human and machine translation. abs/1609.08144, 2016.

[122] R. Yedida and S. Saha. A novel adaptive learning rate scheduler for deep neural networks. *arXiv preprint arXiv:1902.07399*, 2019.

[123] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3320–3328, Cambridge, MA, USA, 2014. MIT Press.

[124] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[125] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, NSDI '17, 2017.

[126] J. Zhang, H.-F. Yu, and I. S. Dhillon. Autoassist: A framework to accelerate training of deep neural networks. *ArXiv*, abs/1905.03381, 2019.

[127] Z. Zhang, P. Luo, C. C. Loy, and X. Tang. Facial landmark detection by deep multi-task learning. In *ECCV*, 2014.

[128] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[129] S. Zilberstein. Using anytime algorithms in intelligent systems. In *AI Magazine, 17(3):73-83*, 1996.