# From Compute to Data:
# Across-the-Stack System Design for Intelligent Applications

by

Yiping Kang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Assistant Professor Lingjia Tang, Co-Chair
Assistant Professor Jason O. Mars, Co-Chair
Assistant Professor Hun-Seok Kim
Professor Trevor N. Mudge

Yiping Kang

ypkang@umich.edu

ORCID iD: 0000-0002-5964-3655

*To my mother Cen Zhu and father Jie Kang.*

# ACKNOWLEDGEMENTS

It was a challenging and fulfilling journey getting to where I am today and it would not have been possible without all the wise and kind people that have helped me along the way. Jason and Lingjia, you make a dream team of advisors and I can't thank you enough for how much I've learnt from you. Lingjia, your extraordinarily high standard for quality drives me to always demand the absolutely best out of myself and in turn I am able to be proud of my work. Jason, your unparalleled passion for truly impactful research have pushed me to take risks to tackle the most challenging problems. Trev, I am grateful for the opportunity to start pursuing my passion for research as an undergraduate student and navigate my way to a dedicated researcher under your guidance. To my dissertation committee, I am thankful for your insights and advice in this final stage of my studies. I would be remiss if I didn't acknowledge my colleagues. I have forged some of my best friendships as part of TronLab and Clarity Lab and thank you for helping me learn and grow as a scientist.

To my parents, Cen Zhu and Jie Kang, thank you for supporting me through my entire academic career. It is your commitment and sacrifices that made it possible for my dream to come true and I am forever grateful for that. You taught me, by example, that gratitude reciprocates and that integrity is a person's most valuable trait. You are the two people in the world that I want to make the most proud of me. To all my family and friends, from the bottom of my heart, thank you.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# ABSTRACT

Intelligent applications such as Apple Siri, Google Assistant and Amazon Alexa have gained tremendous popularity in recent years. With human-like understanding capabilities and natural language interface, this class of applications is quickly becoming people's preferred way of interacting with their mobile, wearable and smart home devices. There have been considerable advancement in machine learning research that aim to further enhance the understanding capability of intelligent applications, however there exist significant roadblocks in applying state-of-the-art algorithms and techniques to a real-world use case. First, as machine learning algorithms becomes more sophisticated, it imposes higher computation requirements for the underlying software and hardware system to process intelligent application request efficiently. Second, state-of-the-art algorithms and techniques is not guaranteed to provide the same level of prediction and classification accuracy when applied to tasks required in real-world intelligent applications, which are often different and more complex than what are studied in a research environment.

This dissertation addresses these roadblocks by investigating the key challenges across multiple components in an intelligent application system. Specifically, we identify the key compute and data challenges and presents system design and techniques. To improve the computational performance of the hardware and software system, we challenge the status-quo approach of cloud-only intelligent application processing and propose computation partitioning strategies that effectively leverage both the cycles in the cloud and on the mobile device to achieve low latency, low energy consump-

tion and high datacenter throughput. We characterize and taxonomize state-of-the-art deep learning based natural language processing (NLP) applications to identify the algorithmic design elements and computational patterns that render conventional GPU acceleration techniques ineffective on this class of applications. Leveraging their unique characteristics, we design and implement a novel fine-grain cross-input batching techniques for providing GPU acceleration to a number of state-of-the-art NLP applications. For the data component, large scale and effective training data, in addition to algorithm, is necessary to achieve high prediction accuracy. We investigate the challenge of effective large-scale training data collection via crowdsourcing. We propose novel metrics to evaluate the quality of training data for building real-word intelligent application systems. We leverage this methodology to study the trade-off of multiple crowdsourcing methods and provide recommendations on best training data crowdsourcing practices.

# CHAPTER I

# Introduction

Intelligent applications have become increasingly knowledgeable and capable and they are the default way for many people to interact with their personal computing devices. These applications take natural input such as voice and images and have human-like understanding capability to analyze these inputs and provide user with an intelligent response. Intelligent applications leverage state-of-the-art AI algorithms and techniques and there have been considerable advancement in this research area to further boost the capabilities of this class of applications. However, there exists significant roadblocks when it comes to applying these research advancement to real-world use cases. The first roadblock is computational performance. As the machine learning algorithms evolve to leverage more sophisticated computation, the system needs to be optimized for efficiently processing these requests to achieve low response latency and energy consumption. The second roadblock is accuracy. Algorithms and techniques that are proven to achieve state-of-the-art accuracy on research tasks are not guaranteed to provide the same level of high prediction and classification accuracy when solving problems required for real-world use cases.

To remove these roadblocks, we need to study the three major components in an intelligent application system, specifically, algorithm, compute and data. The AI research community and industry have made massive stride on algorithms and

have converged that Deep Neural Network (DNN) is the algorithm of the future as it has been proven to provide state-of-the-art accuracy on the most challenging tasks required in an intelligent applications across multiple domains. There have been a series of effort to provide acceleration for intelligent applications and DNN but the majority of the prior work focus on accelerating one class of neural network in the datacenter, leaving computation cycles on the mobile devices idle and it's also unclear how these techniques could apply to evolving neural network topology. In addition to algorithm, large scale and high quality training data is crucial in building a highly accurate machine learning model yet it's still largely an open research question as to what's the most effective data collection and curation process.

This dissertation, from compute to data, identify the key challenges in leveraging state-of-the-art machine learning algorithms and techniques to build intelligent applications for real-world use cases and present system design and techniques to address these challenges.

## 1.1 Motivation

This section motivates the need for an across-the-stack system design for intelligent application in the context of the key challenges exist in the compute and data aspect.

### 1.1.1 Intelligent Applications Computation

Processing speech and image inputs for Intelligent applications requires accurate and highly sophisticated machine learning techniques, the most common of which are Deep Neural Networks (DNNs). DNNs have become increasingly popular as the core machine learning technique in these applications due to their ability to achieve high accuracy for tasks such as speech recognition, image classification and natural language understanding. Many companies, including Google, Microsoft, Facebook, and Baidu, are using DNNs as the machine learning component for numerous applications

in their production systems [3, 8, 10].

Prior work has shown that speech or image queries for DNN-based intelligent applications require orders of magnitude more processing than text based inputs [42]. The common wisdom has been that traditional mobile devices cannot support this large amount of computation with reasonable latency and energy consumption. Thus, the status quo approach used by web service providers for intelligent applications has been to host all the computation on high-end cloud servers [1, 2, 9, 15]. Queries generated from a user's mobile device are sent to the cloud for processing. However, with this approach, large amounts of data (e.g., images, video and audio) are uploaded to the server via the wireless network, resulting in high latency and energy costs. While data transfer becomes the latency and energy bottleneck, performance and energy efficiency of modern mobile hardware have continued to improve through powerful mobile SoC integration [17, 40]. With this shifting paradigm, several key research questions arise:

1. How feasible it is to execute large-scale intelligent workloads on today's mobile platforms?

2. At what point is the cost of transferring speech and image data over the wireless network too high to justify cloud processing?

3. What role should the mobile edge play in providing processing support for intelligent applications requiring heavy computation?

### 1.1.2 Evolving Algorithms

This proliferation of intelligent applications and deep learning algorithms has been accompanied by a surge in the volume of literature focused on accelerating deep learning computation, which has focused almost exclusively on deep neural networks

with computational patterns that are statically-defined and predictable for each input [23, 25, 27, 36, 43, 58, 65, 69].

To continue to make strides in application accuracy, state-of-the-art deep learning approaches are evolving toward sophisticated, complex algorithms where the actions of the algorithm and the underlying computational patterns depend heavily on the nature of the input, which dynamically influences the structure and dependencies within the computation. This fundamentally contrasts with conventional neural network designs, where the neural network computation and invocations are fixed. This shift has impacted image processing [56], but has been particularly evident in the domain of natural language processing (NLP), where algorithms with complex tree-structures are being leveraged to provide state-of-the-art accuracy on problems that include language understanding, sentiment analysis, and question answering [45, 79, 80]. The computation of this emerging class of intelligent applications are dynamically defined at runtime based on the structure and content of its input query.

The dramatic difference between the algorithmic composition of conventional deep learning techniques gives rise to a number of important research questions for architects and system designers who seek to support widely proliferating intelligent applications:

1. Are NLP deep learning algorithms as amenable to acceleration on GPUs as deep learning algorithms for other intelligent application domains?

2. What are the characteristics and computational patterns of state-of-the-art deep learning based NLP applications that make them more or less suitable for conventional GPU acceleration?

3. How should system designs be changed and updated to reflect the increasing complexity of deep learning algorithms for NLP?

### 1.1.3 Data Collection

Large, high quality corpora are crucial in the development of effective machine learning models in many areas. The performance of the machine learning models, especially deep learning models, depend heavily on the quantity and quality of the training data. Developing intelligent applications such as Apple Siri, Google Assistant and Amazon Alexa poses a significant challenge for data collection as we need to do rapid prototyping and bootstrapping to train new intelligent application capabilities. The use of crowdsourcing has enabled the creation of large corpora at relatively low cost [78] and is critical in collecting the quantities of data required to train models with high accuracy. However, designing effective methodologies for data collection with the crowd is largely an open research question [74].

There exists a major challenge when collecting data to build a real-world intelligent application. We have observed that the complexity of building dialogue system for a real-world use case is often substantially greater than those studied in the research community. Therefore, a large amount of high quality training data tailored to the target problem is critical for creating the best user experience in a real-world intelligent application.

Crowdsourcing offers a promising solution by massively parallelizing data collection efforts across a large pool of workers at relatively low cost. Because of the involvement of crowd workers, collecting high-quality data efficiently requires careful orchestration of crowdsourcing jobs, including their instructions and prompts. In order to collect the large-scale tailored dataset we need via crowdsourcing, there are several research questions we need to answer:

- How can we evaluate the effectiveness of crowdsourcing methods and the quality of the datasets collected via these methods?

- During the data collection process, how can we identify the point when addi-

tional data would have diminishing returns on the performance of the downstream trained models?

- Which crowdsourcing method yields the highest-quality training data for intent classification in a real-world intelligent dialogue system?

## 1.2 Across-the-Stack System Design for Intelligent Applications

This sections summarizes system design and techniques optimizing the compute and data component of an intelligent application.

### 1.2.1 Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge

This dissertation first presents an investigation into the current cloud-only approach of intelligent application processing to better understand the bottleneck of end-to-end intelligent application query.

Based on our investigation using 8 DNN-based intelligent applications spanning the domains of vision, speech, and natural language, we discover that, for some applications, due to the high data transfer overhead, locally executing on the mobile device can be an order of magnitude faster than the cloud-only approach.

Furthermore, we find that instead of limiting the computation to be either executed entirely in the cloud or entirely on the mobile, a fine-grained layer-level partitioning strategy based on a DNN's topology and constituent layers can achieve far superior end-to-end latency performance and mobile energy efficiency. By pushing compute out of the cloud and onto the mobile devices, we also improve datacenter throughput, allowing a given datacenter to support many more user queries, and creating a win-win situation for both the mobile and cloud systems.

Given the observation that ideal fine-grained DNN partition points depend on the layer compositions of the DNN, the particular mobile platform used, the wireless network configuration and the server load, we design a lightweight dynamic scheduler, `Neurosurgeon`. `Neurosurgeon` is a runtime system spanning cloud and mobile platforms that automatically identifies the ideal partition points in DNNs and orchestrates the distribution of computation between the mobile device and the datacenter. `Neurosurgeon` partitions the DNN computation and takes advantage of the processing power of both the mobile and the cloud while reducing data transfer overhead.

### 1.2.2 Accelerating Deep Learning Based Natural Language Processing Applications

We taxonomize the behavior of the state-of-the-art deep learning based Natural Language Processing applications, finding that there are fundamental algorithmic patterns and computational characteristics that cause this class to map poorly to conventional acceleration techniques. Specifically, this class of algorithms is characterized by a large number of recurring computations, where 1) each is dependent on previous such computations in a manner defined by the specifics of the problem input (e.g., semantic structure of a sentence), and 2) each takes the form of a small version of conventional neural network computation. Contrary to conventional wisdom on mapping deep learning computation to GPU accelerators, which holds that deep learning computation can be accelerated as long as there is enough computation per byte moved, we find that two additional factors impact accelerability in this new class of applications: the number of neural network kernel invocations and the nature of the dependencies between invocations. In particular, because current techniques [43] to provide high-throughput DNN services on GPUs rely on the assumption that each input involves the same amount of DNN invocations (often only 1 for im-

age/vision workloads) and the type of DNN computation involved in each input is statically determined with no inter-dependencies, these heretofore unseen characteristics make current techniques unsuitable for accelerating dynamically-structured NLP deep learning applications.

Building on these insights, we introduce *fine-grained cross-input batching*(FGCIB), a novel batching technique that addresses the limitations on conventional batching approaches used in prior work to accelerate conventional deep learning computations. FGCIB works by forming fine-grain, cross-input batches of neural network computation to provide sufficient work to the GPU to achieve high GPU occupancy. Our technique exposes a new form a cross-query parallelism, allowing the execution of multiple queries by breaking the inherent computational dependency within a query to aggregate work across different parts of different queries.

### 1.2.3 Data Collection for Real-World Intelligent Application

We study the process of collecting large scale training data via crowdsourcing and optimize the effectiveness of the process. There is limited work on effective techniques to evaluate a crowdsourcing method and the data collected using that method. Prior work has focused on intrinsic analysis of the data, lacking quantitative investigation of the data's impact on downstream model performance [47]. In this dissertation, we propose two novel metrics to evaluate dataset quality. Specifically, we introduce (1) `coverage`, quantifying how well a training set covers the expression space of a certain task, and (2) `diversity`, quantifying the heterogeneity of sentences in the training set. We focus on one aspect of an intelligent application system, intent classification. We verify the effectiveness of both metrics by correlating them with the model accuracy of two well-known algorithms, SVM [34] and FastText [20,48]. We show that while `diversity` gives a sense of the variation in the data, `coverage` closely correlates with the model accuracy and serves as an effective metric for evaluating

training data quality.

We then describe in detail two crowdsourcing methods to collect intent classification data for building real-world intelligent application. The key ideas of these two methods are (1) describing the intent as a scenario or (2) providing an example sentence to be paraphrased. We experiment multiple variants of these methods by varying the number and type of prompts and collect training data using each variant. We perform metric and accuracy evaluation of these datasets and show that using a mixture of different prompts and sampling paraphrasing examples from real user queries yield training data with higher `coverage` and `diversity` and lead to better performing models.

## 1.3 Summary of Contributions

This dissertation presents a host of system design and techniques to address challenges in building an effective intelligent application for real-world use cases. This section summarizes the specific contributions.

- **DNN computation partitioning across the cloud and mobile edge** – We provide an in-depth layer-level characterization of the compute and data size of 8 DNNs spanning across computer vision, speech and natural language processing. Our investigation reveals that DNN layers have significantly different compute and data size characteristics depending on their type and configurations. Based on this observation, we show that partitioning DNN at layer granularity offers significant performance benefits. We then design `Neurosurgeon`, a system to intelligently partition DNN computation between the mobile and cloud. We demonstrate that `Neurosurgeon` significantly improves end-to-end latency, reduces mobile energy consumption, and improves datacenter throughput.

- **Accelerating state-of-the-art NLP applications** – We characterize the al-

gorithm and computational patterns of a suite of state-of-the-art deep learning based NLP applications and identify a set of unique characteristics that are drastically different from conventional deep learning applications. We demonstrate that the current state-of-the-art technique is not suitable for these emerging class of applications. Based on the insights we gained, we then design a novel batching technique that forms fine-grain batches of neural network computation across multiple queries to efficiently to sustain high GPU occupancy to accelerate NLP applications with complex algorithmic and computational patterns. We use an industry-grade load generator and perform a real-system end-to-end evaluation to demonstrate our system achieves significant throughput and latency improvement over the existing GPU-based deep learning acceleration system.

- **Systematic data collection process** – We introduce two novel metrics to evaluate the quality of training dataset. Specifically, we introduce `diversity` to capture the semantic heterogeneity in the training data and `coverage` to evaluate the effectiveness of a training dataset at representing the target task. We validate the metrics by showing their correlation with the accuracy of the downstream trained models. We then describe two popular crowdsourcing methods for collecting training data for intelligent dialogue system and variants of these two methods. We leverage the proposed metrics to evaluate the quality of the training data collected via these methods. We observe that using crowdsourcing prompts based on real user queries and including a mixture of generic and specific prompts yields training data with high quality.

# CHAPTER II

# Background and Related Work

This section introduces the background on the intelligent applications and machine learning algorithms studied in this dissertation as well as related literature on system design for intelligent applications.

## 2.1 Intelligent Applications

### 2.1.1 Deep Neural Networks

In this section, we provide an overview of Deep Neural Network (DNN) and describe how computer vision, speech, and natural language processing applications leverage DNNs as their core machine learning algorithm.

DNNs are organized in a directed graph where each node is a processing element (a neuron) that applies a function to its input and generates an output. Figure 2.1 depicts a 5 layer DNN for image classification where computation flows from left to right. The edges of the graph are the connections between each neuron defining the flow of data. Multiple neurons applying the same function to different parts of the input define a layer. For a forward pass through a DNN, the output of a layer is the input to the next layer. The depth of a DNN is determined by the number of layers. Computer Vision (CV) applications use DNNs to extract features from an input

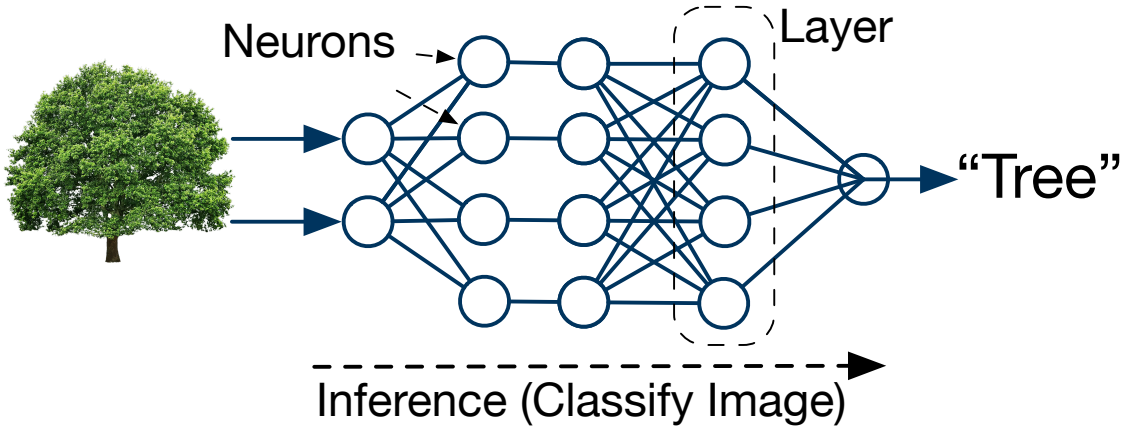



**Figure 2.1: A 5-layer Deep Neural Network (DNN) classifies input image into one of the pre-defined classes.**

image and classify the image into one of the pre-defined classes. Automatic Speech Recognition (ASR) applications use DNNs to generate predictions for speech feature vectors, which will then be post-processed to produce the most-likely text transcript. Natural Language Processing (NLP) applications use DNNs to analyze and extract semantic and syntactic information from word embedding vectors generated from input text.

### 2.1.2 State-of-the-art Natural Language Processing Applications

Recent advances in machine learning techniques has prompted the emergence of applications where users interact with their personal computing devices using natural language rather than a constrained set of buttons and fields. The category of machine learning tasks facilitating this transition, natural language processing (NLP), has become critical to the evolution of modern user interfaces. In this work, we aim to answer research questions as system designers building datacenter systems hosting state-of-the-art NLP applications. We aim to study NLP applications that are 1) representative of complete applications designed to service user queries and 2) achieve the state-of-the-art accuracy in solving their respective tasks. Based on these criteria, we surveyed the recent publications and select 3 applications solving two of the most

prominent problems among the NLP community: *sentiment analysis* and *automatic text summarization*.

**Sentiment Analysis** Sentiment Analysis analyzes the emotions and attitudes in natural language, an application that plays a pivotal role in business planning, political campaigns, and social media analysis. [57,75]. In this work, we investigate a convolutaional neural network based implementation [50] (CNN) and a tree-structured long short-term memory neural network based implementation [80] (LSTM). CNN and LSTM achieve state-of-the-art accuracy on binary and 5-class sentiment analysis, respectively.

**Summarization** Automatic Text Summarization extracts the crux from a body of text, allowing users and higher-level algorithms to ignore extraneous information. Automatic summarization is widely used in news and content delivery services, for example by news agency and websites to automatically generate synopses, keywords and titles of news articles [4,7]. In this work, we study the abstractive summarization application, NAMAS [73] which is designed at Facebook to generate news title based on the first sentence of a news article.

## 2.2 Related Work

### 2.2.1 Computation Partitioning

Previous research efforts focus on offloading computation from the mobile to cloud. COMET [38] offloads a thread when its execution time exceeds a pre-defined threshold. Odessa [70] makes computation partition decisions based on the execution time and data requirements of part of the function. CloneCloud [28] makes the same offloading decisions for all invocations of the same function. MAUI's [35] makes predictions for each function invocation separately and considers the entire application when choosing which function to offload.

### 2.2.2 Datacenter Systems for Accelerating Intelligent Applications

There has been growing interest in building large scale datacenter systems for Deep Neural Network workloads. Various accelerators, such as GPUs, ASICs, and FPGAs, have been proposed for datacenters to better handle DNN computation [24,42,59,66]. There has also been effort in designing compact DNNs suitable for the mobile edge. Microsoft and Google explore small-scale DNNs for speech recognition on mobile platforms [53,54]. MCDNN [41] proposes generating alternative DNN models to trade-off accuracy for performance/energy and choosing to execute either in the cloud or on the mobile. This work investigates intelligent collaboration between the mobile device and cloud for executing traditionally cloud-only large-scale DNNs for reduced latency and energy consumption without sacrificing the DNNs' high prediction accuracy.

Recent work also investigates deep learning based applications across a spectrum of workloads. Many focus on image based workloads involving CNN as they have fixed topologies and large compute for which custom accelerators are desirable [21, 23,69]. Most recently Fathom [18], a benchmark suite, started looking at deep learning based applications beyond CNN based workload and show the different computational characteristics across different types of applications, including a set of NLP based applications, a promising step in the direction of exploring other types of deep learning based applications.

### 2.2.3 Data Collection and Curation

This study complements a line of work on understanding how to effectively collect data with non-expert workers. The closest work is [47]'s study of a range of interface design choices that impact the quality and diversity of crowdsourced paraphrases. However, their work focused on intrinsic evaluation of the paraphrases only, whereas we explore the impact on performance in a downstream task. The variations we consider are also complementary to the aspects covered by their study, providing

additional guidance for future data collection efforts.

In terms of the variations we consider, the closest work is [72], who also considered how task framing can impact behavior. Their study made a more drastic change than ours though, attempting to shift workers' intrinsic motivation by changing the perspective to be about assisting a non-profit organization. While this shift did have a significant impact on worker behavior, it is often not applicable.

More generally, starting with the work of [78] there have been several investigations of crowdsourcing design for natural language processing tasks. Factors that have been considered include quality control mechanisms [71], payment rates and task descriptions [39], task naming [82], and worker qualification requirements [49]. Other studies have focused on exploring variations for specific tasks, such as named entity recognition [37]. Recent work has started to combine and summarize these observations together into consistent guidelines [74], though the range of tasks and design factors makes the scope of such guidelines large. This dissertation adds to this literature, introducing new metrics and evaluation methods to guide crowdsourcing practice.

# CHAPTER III

# Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge

The computation for today's intelligent personal assistants such as Apple Siri, Google Now, and Microsoft Cortana, is performed in the cloud. This cloud-only approach requires significant amounts of data to be sent to the cloud over the wireless network and puts significant computational pressure on the datacenter. However, as the computational resources in mobile devices become more powerful and energy efficient, questions arise as to whether this cloud-only processing is desirable moving forward, and what are the implications of pushing some or all of this compute to the mobile devices on the edge.

In this chapter, we examine the status quo approach of cloud-only processing and investigate computation partitioning strategies that effectively leverage both the cycles in the cloud and on the mobile device to achieve low latency, low energy consumption, and high datacenter throughput for this class of intelligent applications. Our study uses 8 intelligent applications spanning computer vision, speech, and natural language domains, all employing state-of-the-art Deep Neural Networks (DNNs) as the core machine learning technique. We find that given the characteristics of DNN algorithms, a fine-grained, layer-level computation partitioning strategy based on the data and computation variations of each layer within a DNN has significant

latency and energy advantages over the status quo approach.

Using this insight, we design `Neurosurgeon`, a light-weight scheduler to automatically partition DNN computation between mobile devices and datacenters at the granularity of neural network layers. `Neurosurgeon` does not require per-application profiling. It adapts to various DNN architectures, hardware platforms, wireless networks, and server load levels, intelligently partitioning computation for best latency or best mobile energy. We evaluate `Neurosurgeon` on a state-of-the-art mobile development platform and show that it improves end-to-end latency by 3.1× on average and up to 40.7×, reduces mobile energy consumption by 59.5% on average and up to 94.7%, and improves datacenter throughput by 1.5× on average and up to 6.7×.

## 3.1 Cloud-only Processing: The Status Quo

Currently, the status quo approach used by cloud providers for intelligent applications is to perform all DNN processing in the cloud [1, 2, 9, 15]. A large overhead of this approach is in sending data over the wireless network. In this section, we investigate the feasibility of executing large DNNs entirely on a state-of-the-art mobile device, and compare with the status quo.

### 3.1.1 Experimental setup

We use a real hardware platform, representative of today's state-of-the-art mobile devices, the Jetson TK1 mobile platform developed by NVIDIA [12] and used in the Nexus 9 tablet [13]. The Jetson TK1 is equipped with one of NVIDIA's latest mobile SoC, Tegra K1: a quad-core ARM A15 and a Kepler mobile GPU with a single streaming multiprocessor (Table 3.1). Our server platform is equipped with an NVIDIA Tesla K40 GPU, one of NVIDIA's latest offering in server class GPUs (Table 3.2).

**Table 3.1: Mobile Platform Specifications**

| Hardware | Specifications |
|---|---|
| System | Tegra K1 SoC |
| CPU | 4-Plus-1 quad-core ARM Cortex A15 CPU |
| Memory | 2 GB DDR3L 933MHz |
| GPU | NVIDIA Kepler with 192 CUDA Cores |

**Table 3.2: Server Platform Specifications**

| Hardware | Specifications |
|---|---|
| System | 4U Intel Dual CPU Chassis, 8×PCIe 3.0×16 slots |
| CPU | 2× Intel Xeon E5-2620 V2, 6C, 2.10 GHz |
| HDD | 1TB 2.5" HDD |
| Memory | 16× 16GB DDR3 1866MHz ECC/Server Memory |
| GPU | NVIDIA Tesla K40 M-Class 12 GB PCIe |

We use Caffe [46], an actively developed open-source deep learning library, for the mobile and server platform. For the mobile CPU, we use OpenBLAS [83], a NEON-vectorized matrix multiplication library and use the 4 cores available. For both GPUs, we use cuDNN [26], an optimized NVIDIA library that accelerates key layers in Caffe, and use Caffe's CUDA implementations for rest of the layers.

### 3.1.2 Examining the Mobile Edge

We investigate the capability of the mobile platform to execute a traditionally cloud-only DNN workload. We use AlexNet [51] as our application, a state-of-the-art Convolutional Neural Network for image classification. Prior work has noted that AlexNet is representative of today's DNNs deployed in server environments [29].

In Figure 3.1, we break down the latency of an AlexNet query, a single inference on a 152KB image. For wireless communication, we measure the bandwidth of 3G, LTE, and Wi-Fi on several mobile devices using TestMyNet [14].

**Communication Latency** – Figure 3.1a shows the latency to upload the input image via 3G, LTE, and Wi-Fi. The slowest is 3G connection taking over 870ms. LTE and Wi-Fi connection require 180ms and 95ms to upload, respectively, showing

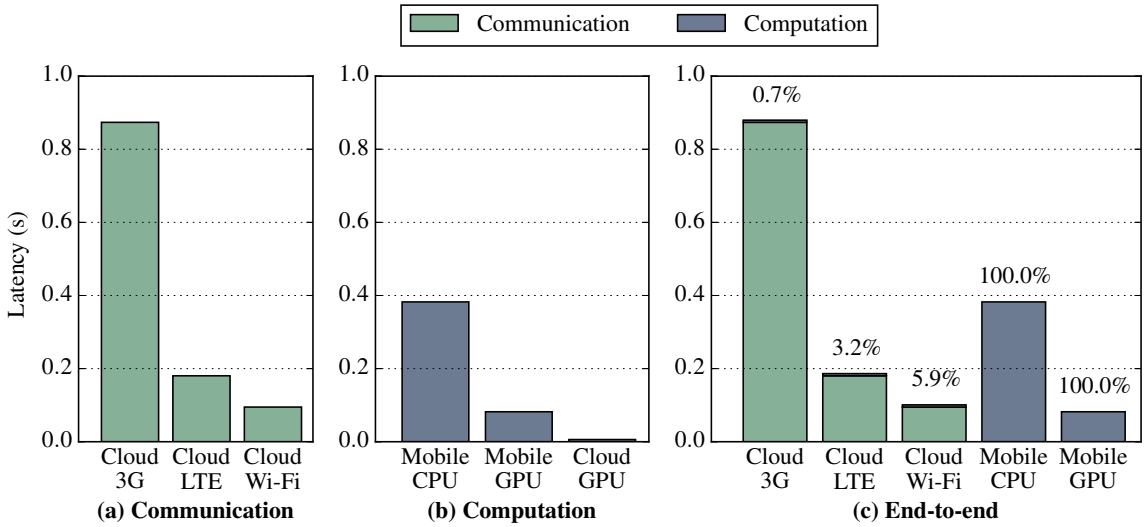



**Figure 3.1: Latency breakdown for AlexNet (image classification). The cloud-only approach is often slower than mobile execution due to the high data transfer overhead.**

that the network type is critical for achieving low latency for the status quo approach.

**Computation Latency** – Figure 3.1b shows the computation latency on mobile CPU, GPU and cloud GPU. The slowest platform is the mobile CPU taking 382ms to process while the mobile GPU and cloud GPU take 81ms and 6ms, respectively. Note that the mobile CPU's time to process the image is still 2.3× faster than uploading input via 3G.

**End-to-end Latency** – Figure 3.1c shows the total latency required by the status quo and the mobile-only approach. Annotated on top of each bar is the fraction of the end-to-end latency spent on computation. The status quo approach spends less than 6% of the time computing on the server and over 94% of the time transferring data. The mobile GPU achieves a lower end-to-end latency than the status quo approach using LTE and 3G, while the status quo approach using LTE and Wi-Fi performs better than mobile CPU execution.

**Energy Consumption** – We measure the energy consumption of the mobile device using a Watts Up? meter [16] and techniques described by Huang et al. [44]. Similar to the trends shown in Figure 3.1a, Figure 3.2a shows that the communication energy

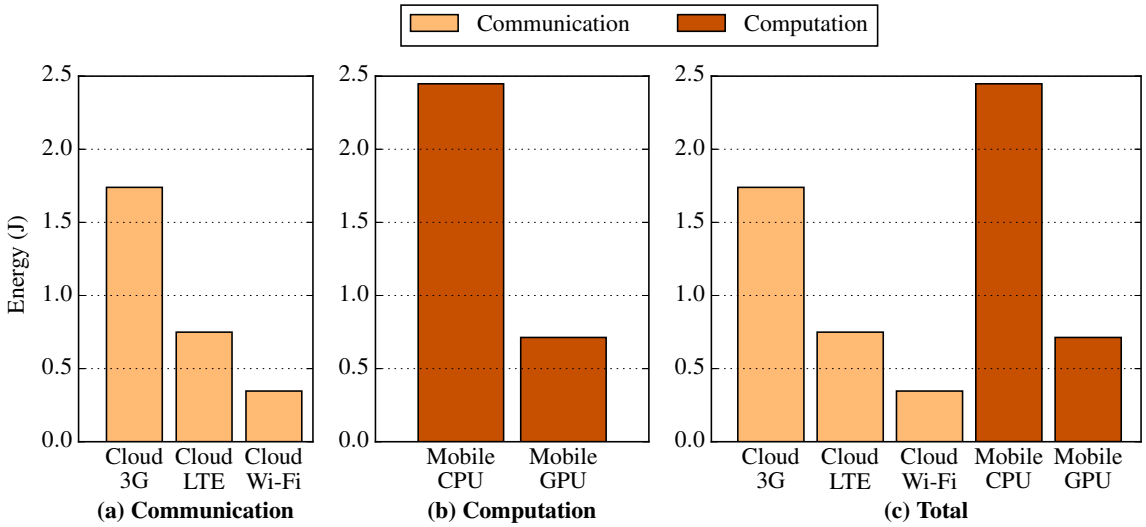



**Figure 3.2: Mobile energy breakdown for AlexNet (image classification). Mobile device consumes more energy transferring data via LTE and 3G than computing locally on the GPU.**

is heavily dependent on the type of wireless network used. In Figure 3.2b, the mobile device's energy consumption is higher on the CPU than the GPU (while the GPU needs more power, the device is used for a shorter burst thus it consumes less total energy). Figure 3.2c shows the total mobile energy consumption for the cloud-only approach and mobile execution where the energy in the cloud-only approach is dominated by communication. The mobile GPU consumes less energy than transferring input via LTE or 3G for cloud processing, while cloud processing via Wi-Fi consumes less energy than mobile execution.

***Key Observations*** – 1) The data transfer latency is often higher than mobile computation latency, especially on 3G and LTE. 2) Cloud processing has a significant computational advantage over mobile processing, but it does not always translate to end-to-end latency/energy advantage due to the dominating data transfer overhead. 3) Local mobile execution often leads to lower latency and energy consumption than the cloud-only approach, while the cloud-only approach achieves better performance if using fast Wi-Fi connection.

## 3.2 Fine-grained Computation Partitioning

Based on the findings in Section 3.1, the question arises as to whether it is advantageous to partition DNN computation between the mobile device and cloud. Based on the observation that DNN layers provide an abstraction suitable for partitioning computation, we begin with an analysis of the data and computation characteristics of state-of-the-art DNN architectures at the layer granularity.

### 3.2.1 Layer Taxonomy

Before the layer-level analysis, it is important to understand the various types of layers present in today's DNNs.

**Fully-connected Layer (`fc`)** – All the neurons in a fully-connected layer are exhaustively connected to all the neurons in the previous layer. The layer computes the weighted sum of the inputs using a set of learned weights.

**Convolution & Local Layer (`conv`, `local`)** – Convolution and local layers convolve the image with a set of learned filters to produce a set of feature maps. These layers mainly differ in the dimensions of their input feature maps, the number and size of their filters, and the stride with which the filters are being applied.

**Pooling Layer (`pool`)** – Pooling layers apply a pre-defined function (e.g., max or average) over regions of input feature maps to group features together. These layers mainly differ in the dimension of their input, size of the pooling region, and the stride with which the pooling is applied.

**Activation Layer** – Activation layers apply a non-linear function to each of its input data individually, producing the same amount of data as output. Activation layers present in the neural networks studied in this work include sigmoid layer (`sig`), rectified-linear layer (`relu`), and hard Tanh layer (`htanh`).

Other layers studied in this work include: **normalization layer (`norm`)** nor-

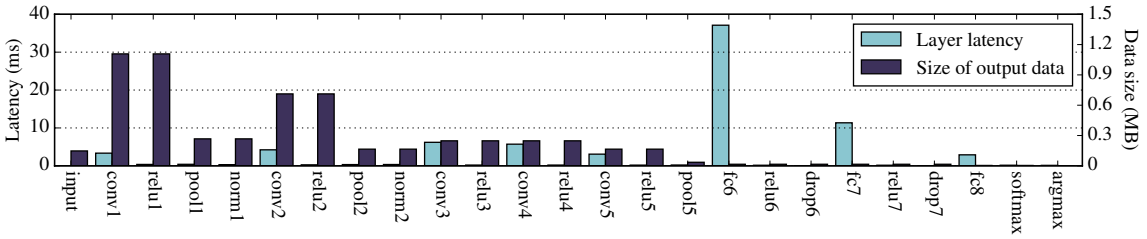



**Figure 3.3: The per layer execution time (the light-colored left bar) and size of data (the dark-colored right bar) after each layer's execution (input for next layer) in AlexNet. Data size sharply increases then decreases while computation generally increases through the network's execution.**

malizes features across spatially grouped feature maps; **softmax layer (`softmax`)** produces a probability distribution over the number of possible classes for classification; **argmax layer (`argmax`)** chooses the class with the highest probability; and **dropout layer (`dropout`)** randomly ignores neurons during training to avoid model over-fitting and are passed through during prediction.

### 3.2.2 Characterizing Layers in AlexNet

We first investigate the data and computation characteristics of each layer in AlexNet. These characteristics provide insights to identify a better computation partitioning between mobile and cloud at the layer level. In the remainder of this and subsequent sections, we use the GPU in both mobile and server platforms.

**Per-layer Latency** – The left bars (light-colored) in Figure 3.3 show the latency of each layer on the mobile platform, arranged from left to right in their sequential execution order. The convolution (`conv`) and fully-connected layers (`fc`) are the most time-consuming layers, representing over 90% of the total execution time. Convolution layers in the middle (`conv3` and `conv4`) takes longer to execute than the early convolution layers (`conv1` and `conv2`). Larger number of filters are applied by the convolution layers later in the DNN to progressively extract more robust and representative features, increasing the amount of computation. On the other hand, fully-connected layers are up to one magnitude slower than the convolution layers in

the network. The most time-consuming layer is the layer `fc6`, a fully-connected layer deep in the DNN, taking 45% of the total execution time.

**Data Size Variations** – The right bars (dark-colored) in Figure 3.3 shows the size of each layer's output data, which is also the input to the next layer. The first three convolution layers (`conv1`, `conv2` and `conv3`) generate large amounts of output data (shown as the largest dark bars) as they apply hundreds of filters over their input feature maps to extract interesting features. The data size stays constant through the activation layers (`relu1 - relu5`). The pooling layers sharply reduce the data size by up to 4.7× as they summarize regions of neighboring features by taking the maximum. The fully-connected layers deeper in the network (`fc6 - fc8`) gradually reduce the data size until the softmax layer (`softmax`) and argmax layer (`argmax`) at the end reduce the data to be one classification label.

***Key Observations*** – 1) Depending on its type and location in the network, each layer has a different computation and data profile. 2) The latency of convolution and pooling layers on the mobile GPU are relatively small, while fully-connected layers incur high latency. 3) Convolution and pooling layers are mostly at the front-end of the network, while fully-connected layers are at the back-end. 4) With convolution layers increasing data and then pooling layers reducing data, the front-end layers altogether reduce the size of data gradually. Data size in the last few layers are smaller than the original input. 5) The findings that data size is generally decreasing at the front-end, and per-layer mobile latency is generally higher at the back-end, indicates the unique opportunity for computation partitioning in the middle of the DNN between the mobile and cloud.

### 3.2.3 Layer-granularity Computation Partitioning

The analysis in Section 3.2.2 indicates that there exist interesting points within a neural network to partition computation. In this section, we explore partitioning

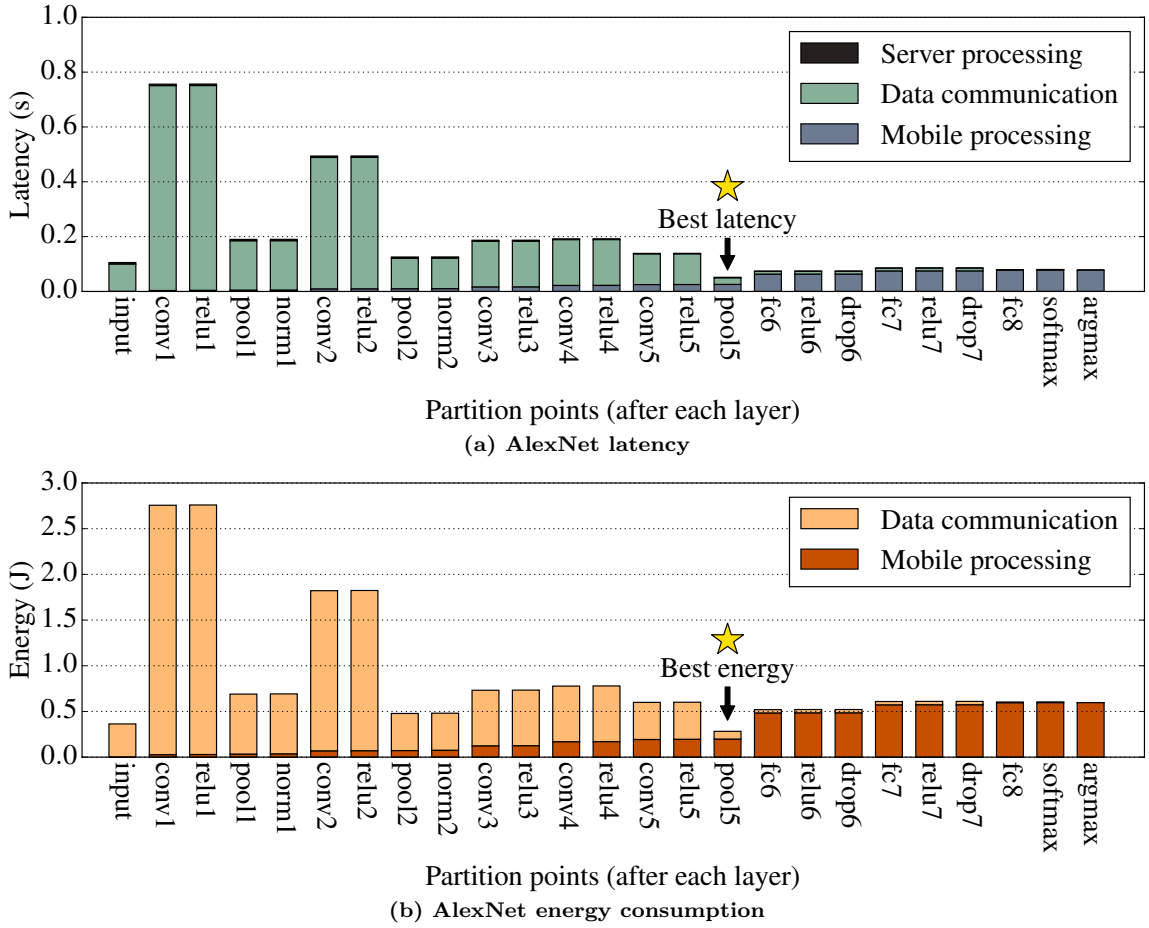



**Figure 3.4: End-to-end latency and mobile energy consumption when choosing different partition points. After the execution of every layer is considered a partition point. Each bar represents the total latency (a) or mobile energy (b) if the DNN is partitioned after the layer marked on the X-axis. The left-most bar represents cloud-only processing and the right-most bar represents mobile-only processing. The partition points for best latency and mobile energy are annotated.**

AlexNet at each layer between the mobile and cloud. In this section, we use Wi-Fi as the wireless network configuration.

Each bar in Figure 3.4a represents the end-to-end latency of AlexNet, partitioned after each layer. Similarly, each bar in Figure 3.4b represents the mobile energy consumption of Alexnet, partitioned after each layer. Partitioning computation after a specific layer means executing the DNN on the mobile up to that layer, transferring the output of that layer to the cloud via wireless network, and executing the remaining layers in the cloud. The leftmost bar represents sending the original input for cloud-

only processing. As partition point moves from left to right, more layers are executed on the mobile device thus there is an increasingly larger mobile processing component. The rightmost bar is the latency of executing the entire DNN locally on the mobile device.

**Partition for Latency** – If partitioning at the front-end, the data transfer dominates the end-to-end latency, which is consistent with our observation in Section 3.2.2 that the data size is the largest at the early stage of the DNN. Partitioning at the back-end provides better performance since the application can minimize the data transfer overhead, while taking advantage of the powerful server to execute the more compute-heavy layers at the back-end. In the case of AlexNet using the mobile GPU and Wi-Fi, partitioning between the last pooling layer (`pool5`) and the first fully-connected layer (`fc6`) achieves the lowest latency, as marked in Figure 3.4a, improving 2.0× over cloud-only processing.

**Partition for Energy** – Similar to latency, due to the high energy cost of wireless data transfer, transferring the input for cloud-only processing is not the most energy-efficiency approach. As marked in Figure 3.4b, partitioning in the middle of the DNN achieves the best mobile energy consumption, 18% more energy efficient than the cloud-only approach.

***Key Observations*** – Partitioning at the layer granularity can provide significant latency and energy efficiency improvements. For AlexNet using the GPU and Wi-Fi, the best partition points are between the intermediate layers of the DNN.

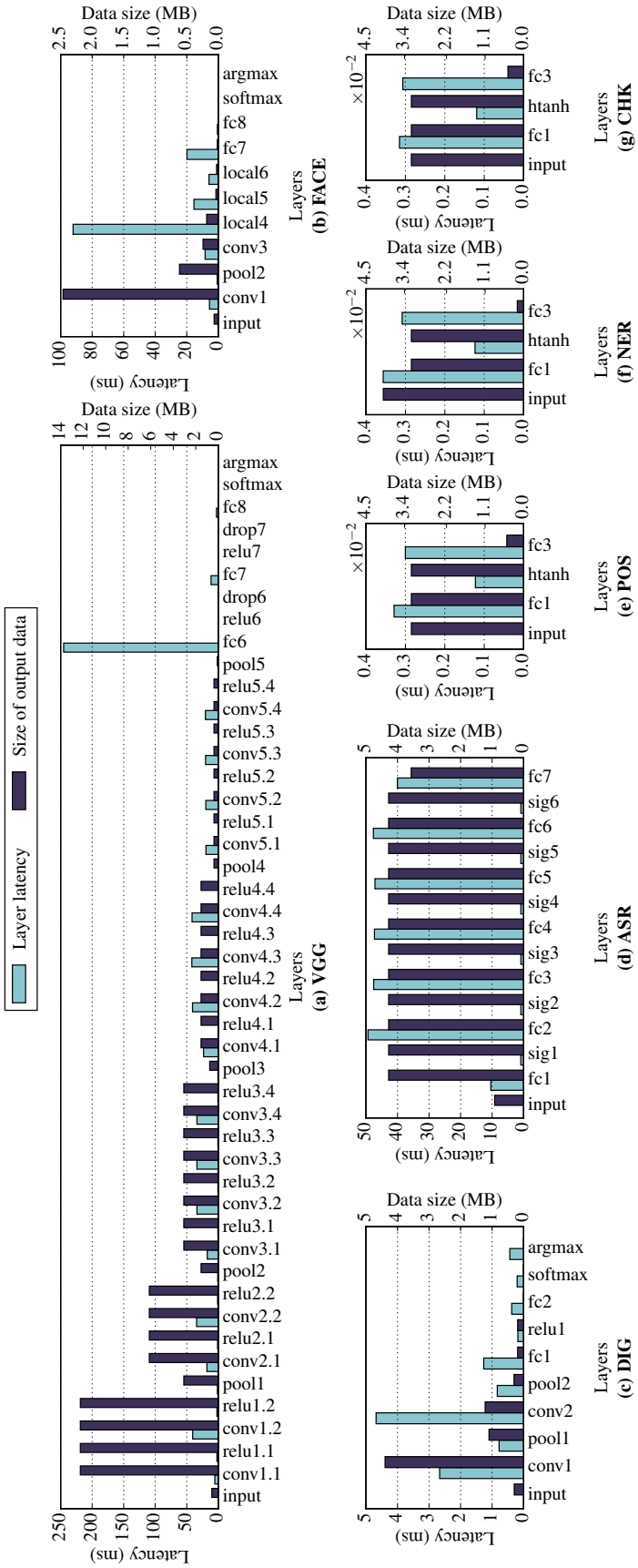



**Figure 3.5: The per layer latency on the mobile GPU (left light-color bar) and size of data (right dark-color bar) after each layer's execution.**

**Table 3.3: Benchmark Specifications**

| App | Abbr. | Network | Input | Layers |
|---|---|---|---|---|
| Image classification | IMC | AlexNet [51] | Image | 24 |
| | VGG | VGG [76] | Image | 46 |
| Facial recognition | FACE | DeepFace [81] | Image | 10 |
| Digit recognition | DIG | MNIST [52] | Image | 9 |
| Speech recognition | ASR | Kaldi [67] | Speech features | 13 |
| Part-of-speech tagging | POS | SENNA [31] | Word vectors | 3 |
| Named entity recognition | NER | SENNA [31] | Word vectors | 3 |
| Word chunking | CHK | SENNA [31] | Word vectors | 3 |

### 3.2.4 Generalizing to more DNNs

We expand our investigation to 7 more intelligent applications to study their data and computation characteristics and their impact on computation partitioning opportunity. We use the DNNs provided in the Tonic suite [42], as well as VGG, a state-of-the-art image classification DNN, and LTE as the wireless network configuration. Details about the benchmarks are listed in Table 3.3. We count the number of layers of each DNN starting from the first non-input layer to the last layer, including `argmax` if present.

**CV Applications** – The three remaining computer vision DNNs (VGG, FACE and DIG) have similar characteristics as AlexNet (Figure 3.3), as shown in Figures 3.5a – 3.5c. The front-end layers are convolution layers increasing data, and pooling layers reducing data. The data size in the back-end layers are similar or smaller than the original input data. The latency for the back-end layers are higher than most of the front-end layers (e.g., `fc6` is the most time-consuming layer in VGG), except for DIG where convolution layers are most time-consuming. Similar to AlexNet, these characteristics indicate partitioning opportunities in the middle of the DNN. Figure 3.6a shows that the partition point for best latency for VGG is in the intermediate layers. In addition, Figures 3.6a - 3.6c show that different CV applications have different partition points for best latency, and Figures 3.7a - 3.7c show the different partition points for best energy for these DNNs.

**ASR and NLP Applications** – The remaining four DNNs in the suite (ASR, POS,

NER and CHK) only consist of fully-connected layers and activation layers. The layer breakdowns are shown in Figures 3.5d - 3.5g, where, throughout the execution, layers of the same type incur similar latency and the data size stay relatively constant except for the very first and last layer of each DNN. These DNNs do not have data-increasing layers (i.e., convolution layers) or data-reducing layers (i.e., pooling layers). As a result, there only exist opportunities for partitioning the computation at the extremities of these networks. Figures 3.6d - 3.6g and Figures 3.7d - 3.7g show the different partition points for best latency and energy for these DNNs, respectively. There are data communication components in the right-most bars (mobile-only processing) for these applications because the output of the DNN is sent to the cloud for post-processing steps required by these applications.

***Key Observations*** – 1) In DNNs with convolution and pooling layers (e.g. Computer Vision applications), the data size increases after convolution layers and decreases after pooling layers, while the per-layer computation generally increases through the execution. 2) DNNs with only fully-connected layers of similar size and activation layers see small variations in per-layer latency and data size (e.g., ASR and NLP DNNs). 3) The best way to partition a DNN depends on its topology and constituent layers. Computer vision DNNs sometimes have better partition points in the middle of the DNN, while it is more beneficial to partition at the beginning or the end for ASR and NLP DNNs. The strong variations in the best partition point suggest there is a need for a system to partition DNN computation between the mobile and cloud based on the neural network architecture.

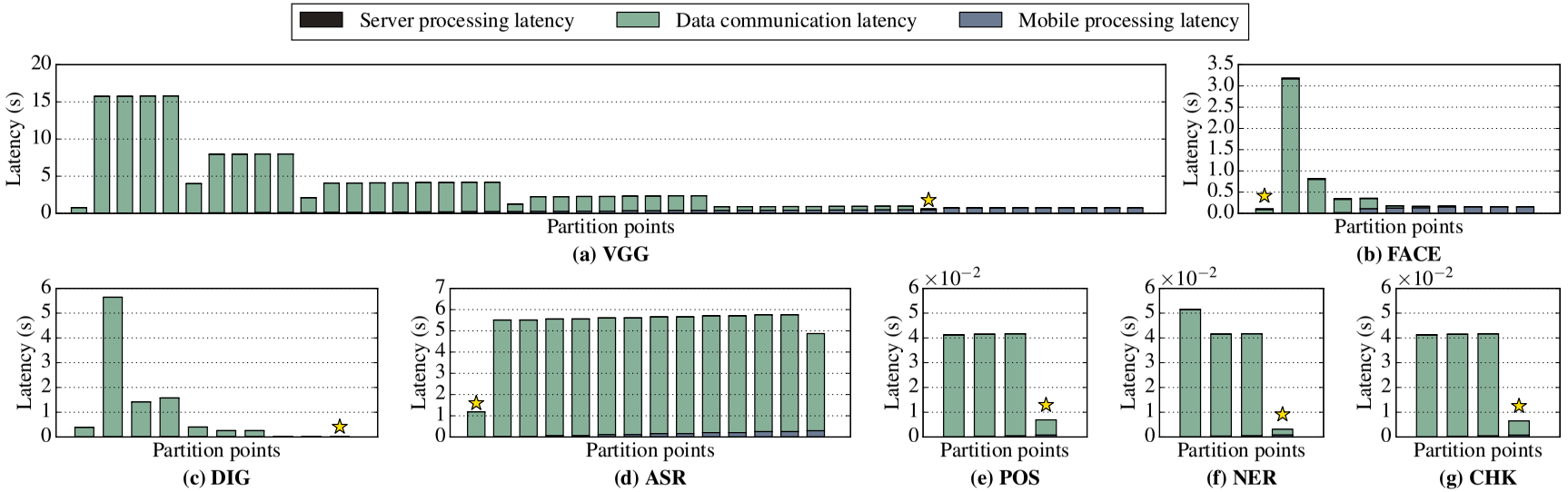



**Figure 3.6: End-to-end latency when choosing different partition points. Each bar represents the end-to-end latency if the DNN is partitioned after each layer, where the left-most bar represents cloud-only processing (i.e., partitioning at the beginning) while the right-most bar represents mobile-only execution (i.e., partitioning at the end). The wireless network configuration is LTE. The partition points for best latency are each marked by ★.**

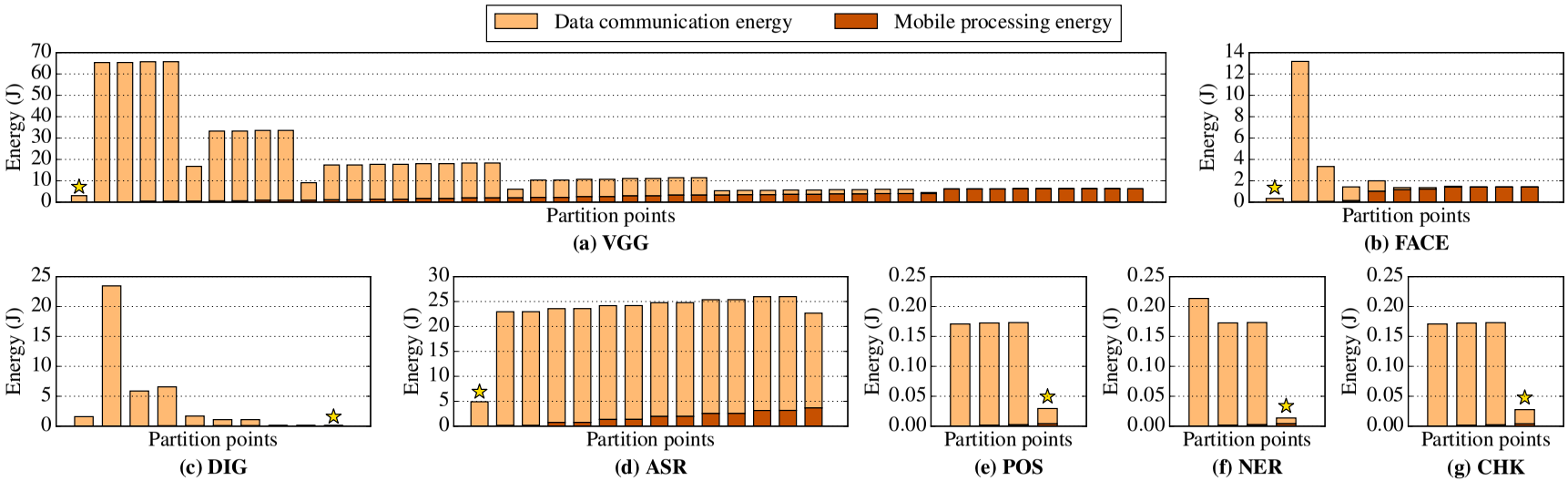



**Figure 3.7: Mobile energy consumption when choosing different partition points. Each bar represents the mobile energy consumption if the DNN is partitioned after each layer, where the left-most bar represents cloud-only processing (i.e., partitioning at the beginning) while the right-most bar represents mobile-only execution (i.e., partitioning at the end). The wireless network configuration is LTE. The partition points for best energy are each marked by ★.**

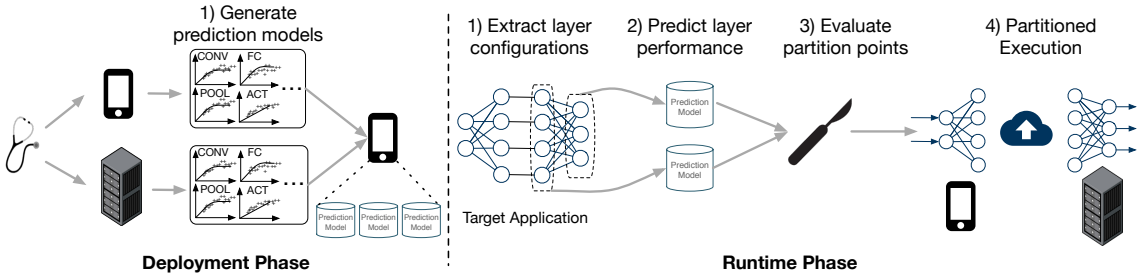



**Figure 3.8: Overview of `Neurosurgeon`. At deployment, `Neurosurgeon` generates prediction models for each layer type. During runtime, `Neurosurgeon` predicts each layer's latency/energy cost based on the layer's type and configuration, and selects the best partition point based on various dynamic factors.**

## 3.3 Neurosurgeon

The best partition point for a DNN architecture depends on the DNN's topology, which manifests itself in the computation and data size variations of each layer. In addition, dynamic factors such as state of the wireless network and datacenter load affect the best partition point even for the same DNN architecture. For example, mobile devices' wireless connections often experience high variances [64], directly affecting the data transfer latency. Datacenters typically experience diurnal load patterns [60], leading to high variance in its DNN query service time. Due to these dynamic factors, there is a need for an automatic system to intelligently select the best point to partition the DNN to optimize for end-to-end latency or mobile device energy consumption. To address this need, we present the design of `Neurosurgeon`, an intelligent DNN partitioning engine. `Neurosurgeon` consists of a deployment phase and a runtime system that manages the partitioned execution of an intelligent application. Figure 3.8 shows the design of `Neurosurgeon`, which has two stages: deployment and runtime.

**At Deployment** – `Neurosurgeon` profiles the mobile device and the server to generate performance prediction models for the spectrum of DNN layer types (enumerated in Section 3.2.1). Note that `Neurosurgeon`'s profiling is application agnostic and only

needs to be done once for a given set of mobile and server platforms; per-application profiling is not needed. This set of prediction models are stored on the mobile device and later used to predict the latency and energy cost of each layer (Section 3.3.1).

**During Runtime** – During the execution of an DNN-based intelligent application on the mobile device, `Neurosurgeon` dynamically decides the best partition point for the DNN. As illustrated in Figure 3.8, the steps are as follows: 1) `Neurosurgeon` analyzes and extracts the DNN architecture's layer types and configurations; 2) the system uses the stored layer performance prediction models to estimate the latency and energy consumption for executing each layer on the mobile and cloud; 3) with these predictions, combined with the current wireless connection bandwidth and datacenter load level, `Neurosurgeon` selects the best partition point, optimizing for best end-to-end latency or best mobile energy consumption; 4) `Neurosurgeon` executes the DNN, partitioning work between the mobile and cloud.

### 3.3.1 Performance Prediction Model

`Neurosurgeon` models the per-layer latency and the energy consumption of arbitrary neural network architecture. This approach allows `Neurosurgeon` to estimate the latency and energy consumption of a DNN's constituent layers without executing the DNN.

We observe that for each layer type, there is a large latency variation across layer configurations. Thus, to construct the prediction model for each layer type, we vary the configurable parameters of the layer and measure the latency and power consumption for each configuration. Using these profiles, we establish a regression model for each layer type to predict the latency and power of the layer based on its configuration. We describe each layer's regression model variables later in this section. We use GFLOPS (Giga Floating Point Operations per Second) as our performance metric. Based on the layer type, we use either a logarithmic or linear function as

the regression function. The logarithmic-based regression is used to model the performance plateau as the computation requirement of the layer approaches the limit of the available hardware resources.

**Convolution, local and pooling** layers' configurable parameters include the input feature map dimension, number, size and stride of the filters. The regression model for convolution layer is based on two variables: the number of features in the input feature maps, and $(filter\ size/stride)^2 \times (\#\ of\ filters)$, which represents the amount of computation applied to each pixel in the input feature maps. For local and pooling layers, we use the size of the input and output feature maps as the regression model variables.

In a **fully-connected** layer, the input data is multiplied by the learned weight matrix to generate the output vector. We use the number of input neurons and number of output neurons as the regression model variables. **Softmax** and **argmax** layers are handled similarly.

**Activation** layers have fewer configurable parameters compared to other layers because activation layers have a one-to-one mapping between their input data and output. We use the number of neurons as the regression model variable. We apply the same approach to **normalization** layers.

As previously mentioned, it is a one-time profiling step required for each mobile and server hardware platform to generate a set of prediction models. The models enable `Neurosurgeon` to estimate the latency and energy cost of each layer based its configuration, which allows `Neurosurgeon` to support future neural network architectures without additional profiling overhead.

### 3.3.2 Dynamic DNN Partitioning

Utilizing the layer performance prediction models, `Neurosurgeon` dynamically selects the best DNN partition points, as described in Algorithm 1. The algorithm has

two-steps: analysis of the target DNN and partition point selection.

**Analysis of the Target DNN** – `Neurosurgeon` analyzes the target DNN's constituent layers, and uses the prediction models to estimate, for each layer, the latency on mobile and cloud, and power consumption on the mobile. Specifically, at lines 11 and 12 of Algorithm 1, `Neurosurgeon` extracts each layer's type and configuration ($L_i$) and uses the regression models to predict the latency of executing layer $L_i$ on mobile ($TM_i$) and cloud ($TC_i$), while taking into consideration of current datacenter load level ($K$). Line 13 estimates the power of executing layer $L_i$ on the mobile device ($PM_i$) and line 14 calculates the wireless data transfer latency ($TU_i$) based on the latest wireless network bandwidth.

**Partition Point Selection** – `Neurosurgeon` then selects the best partition point. The candidate points are after each layer. Lines 16 and 18 evaluate the performance when partitioning at each candidate point and select the point for either best end-to-end latency or best mobile energy consumption. Because of the simplicity of the regression models, this evaluation is lightweight and efficient.

**Algorithm 1** `Neurosurgeon` DNN partitioning algorithm

1: **Input:**
2: $N$: number of layers in the DNN
3: $\{L_i | i = 1 \cdots N\}$: layers in the DNN
4: $\{D_i | i = 1 \cdots N\}$: data size at each layer
5: $f, g(L_i)$: regression models predicting the latency and power of executing $L_i$
6: $K$: current datacenter load level
7: $B$: current wireless network uplink bandwidth
8: $PU$: wireless network uplink power consumption
9: **procedure** PARTITIONDECISION
10: **for each** $i$ *in* $1 \cdots N$ **do**
11: $TM_i \leftarrow f_{mobile}(L_i)$
12: $TC_i \leftarrow f_{cloud}(L_i, K)$
13: $PM_i \leftarrow g_{mobile}(L_i)$
14: $TU_i \leftarrow D_i / B$
15: **if** $OptTarget == latency$ **then**
16: **return** $\arg\min_{j=1 \cdots N} (\sum_{i=1}^{j} TM_i + \sum_{k=j+1}^{N} TC_k + TU_j)$
17: **else if** $OptTarget == energy$ **then**
18: **return** $\arg\min_{j=1 \cdots N} (\sum_{i=1}^{j} TM_i \times PM_i + TU_j \times PU)$

### 3.3.3 Partitioned Execution

We prototype `Neurosurgeon` by creating modified instances of Caffe [46] to serve as our mobile-side (`NSmobile`) and server-side (`NSserver`) infrastructures. Through these two variations of Caffe, we implement our client-server interface using Thrift [77], an open source flexible RPC interface for inter-process communication. To allow for flexibility in the dynamic selection of partition points, both `NSmobile` and `NSserver` host complete DNN models, and partition points are enforced by `NSmobile` and `NSserver` runtime. Given a partition decision by `NSmobile`, execution begins on the mobile device and cascades through the layers of the DNN leading up to that partition point. Upon completion of that layer, `NSmobile` sends the output of that layer from the mobile device to `NSserver` residing on the server side. `NSserver` then executes the remaining DNN layers. Upon the completion of the DNN execution, the final result is sent back to `NSmobile` on the mobile device from `NSserver`. Note that there is exactly one partition point within the DNN for which information is sent from the mobile device to the cloud.

## 3.4 Evaluation

We evaluate `Neurosurgeon` using 8 DNNs (Table 3.3) as our benchmarks across Wi-Fi, LTE and 3G wireless connections with both CPU-only and GPU mobile platforms. We demonstrate `Neurosurgeon` achieves significant end-to-end latency and mobile energy improvements over the status quo cloud-only approach (Sections 3.4.1 and 3.4.2). We then compare `Neurosurgeon` against MAUI [35], a well-known computation offloading framework (Section 3.4.3). We also evaluate `Neurosurgeon`'s robustness to variations in wireless network connections (Section 3.4.4) and server load (Section 3.4.5), demonstrating the need for such a dynamic runtime system. Finally, we evaluate the datacenter throughput improvement `Neurosurgeon` achieves

by pushing compute out of the cloud to the mobile device (Section 3.4.6).

### 3.4.1 Latency Improvement

**Table 3.4: `Neurosurgeon`'s partition point selections for best end-to-end latency. Green block indicates `Neurosurgeon` makes the optimal partition choice and white block means a suboptimal partition point is picked. On average, `Neurosurgeon` achieves within 98.5% of the optimal performance.**

| Mobile | Wireless network | Benchmarks | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | IMC | VGG | FACE | DIG | ASR | POS | NER | CHK |
| CPU | Wi-Fi | input | input | input | input | input | fc3 | | |
| | LTE | input | input | input | argmax | input | fc3 | | |
| | 3G | argmax | input | input | argmax | input | fc3 | | |
| GPU | Wi-Fi | pool5 | input | input | argmax | input | fc3 | | |
| | LTE | argmax | argmax | input | argmax | input | fc3 | | |
| | 3G | argmax | argmax | argmax | argmax | input | fc3 | | |

**Partition Point Selection** – Table 3.4 summarizes the partition points selected by `Neurosurgeon` optimizing for latency across the 48 configurations (i.e., 8 benchmarks, 3 wireless network types, mobile CPU and GPU). The green cells indicate when `Neurosurgeon` selects the optimal partition point and achieves the best speedup while the white cells indicate `Neurosurgeon` selects a suboptimal point. `Neurosurgeon` selects the best partition point for 44 out of the 48 configurations. The mispredictions occur because the partition points and its associated performance are very close to one another and thus a small difference in `Neurosurgeon`'s latency prediction shifts the selection. Across all benchmarks and configurations, `Neurosurgeon` achieves latency speedup within 98.5% of optimal speedup.

**Latency Improvement** – Figure 3.9 shows `Neurosurgeon`'s latency improvement over the status quo approach, across the 8 benchmarks on Wi-Fi, LTE, and 3G. Figure 3.9a shows the latency improvement when applying `Neurosurgeon` to a mobile platform equipped with a CPU, and Figure 3.9b shows that of a mobile platform with a GPU. For CV applications, `Neurosurgeon` identifies the best partition points for 20 out of 24 cases and achieves significant latency speedups, especially when the mo-

bile GPU is available. For the NLP applications, `Neurosurgeon` achieves significant latency speedups even when Wi-Fi is available. For ASR, `Neurosurgeon` successfully identifies that it is best to execute the DNN entirely on the server and, therefore `Neurosurgeon` performs similar to the status quo for that particular benchmark. Across all benchmarks and configurations, `Neurosurgeon` achieves a latency speedup of 3.1× on average and up to 40.7× over the status quo approach.

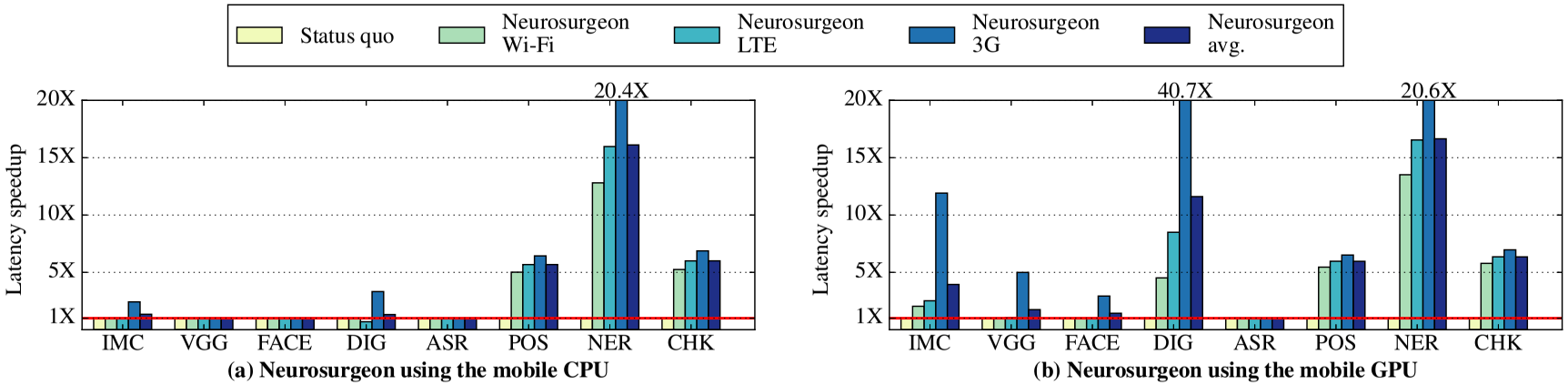



**Figure 3.9: Latency speedup achieved by `Neurosurgeon` normalized to status quo approach (executing entire DNN in the cloud). Results for three wireless networks (Wi-Fi, LTE and 3G) and mobile CPU and GPU are shown here. `Neurosurgeon` improves the end-to-end DNN inference latency by 3.1× on average (geometric mean) and up to 40.7×.**

### 3.4.2 Energy Improvement

**Table 3.5: `Neurosurgeon` partition point selections for best mobile energy consumption. Green block indicates `Neurosurgeon` makes the optimal partition choice and white block means a suboptimal partition point is picked. On average, `Neurosurgeon` achieves a mobile energy reduction within 98.8% of the optimal reduction.**

| Mobile | Wireless network | Benchmarks | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | IMC | VGG | FACE | DIG | ASR | POS | NER | CHK |
| CPU | Wi-Fi | input | input | input | input | input | fc3 | | |
| | LTE | input | input | input | input | input | fc3 | | |
| | 3G | input | input | input | argmax | input | fc3 | | |
| GPU | Wi-Fi | input | input | input | argmax | input | fc3 | | |
| | LTE | pool5 | input | input | argmax | input | fc3 | | |
| | 3G | argmax | argmax | input | argmax | input | fc3 | | |

**Partition Point Selection** – Table 3.5 summarizes the partition points identified by `Neurosurgeon` for best mobile energy. `Neurosurgeon` selects the best partition point for 44 out of the 48 configurations. For the suboptimal choices, `Neurosurgeon` consumes 24.2% less energy on average than the status quo approach.

**Energy Improvement** – Figure 3.10 shows the mobile energy consumption achieved by `Neurosurgeon`, normalized to the status quo approach. Figure 3.10a and 3.10b present results for CPU-only mobile platform and GPU-equipped mobile platform, respectively. When optimizing for best energy consumption, `Neurosurgeon` achieves on average a 59.5% reduction in mobile energy and up to 94.7% reduction over the status quo. Similar to the improvement for latency, the energy reduction is also higher for most benchmarks when the mobile platform is equipped with a GPU.

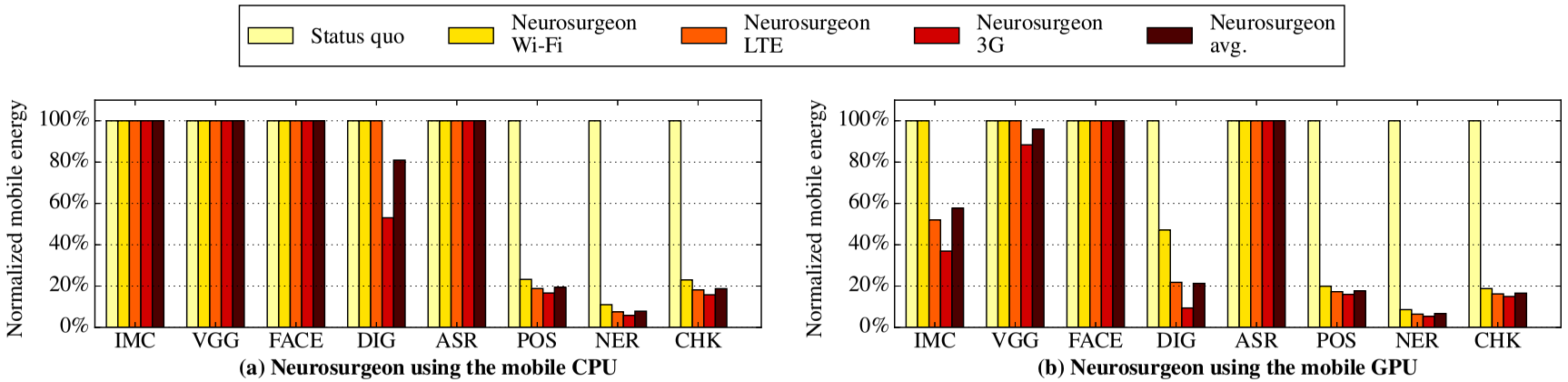



**Figure 3.10: Mobile energy consumption achieved by `Neurosurgeon` normalized to status quo approach (executing entire DNN in the cloud). Results for three wireless networks (Wi-Fi, LTE and 3G) and mobile CPU and GPU are shown here. `Neurosurgeon` reduces the mobile energy consumption by 59.5% on average (geometric mean) and up to 94.7%.**

### 3.4.3 Comparing Neurosurgeon to MAUI

In this section, we compare `Neurosurgeon` to MAUI [35], a general offloading framework. Note that MAUI is control-centric, reasoning and making decisions about regions of code (functions), whereas `Neurosurgeon` is data-centric, making partition decisions based on the structure of the data topology that can differ even if the same code region (function) is called.

Figure 3.11 presents the latency speedup achieved by `Neurosurgeon` normalized to MAUI when executing the 8 DNN benchmarks, averaged across three wireless network types. Figure 3.11a presents the result when applying MAUI and `Neurosurgeon` on a CPU-only mobile platform and Figure 3.11b presents the result on a mobile platform equipped with a GPU. In this experiment, we assume that for MAUI, programmers have optimally annotated the minimal program states that need to be transferred.

Figure 3.11 shows that `Neurosurgeon` significantly outperforms MAUI on the computer vision applications. For the NLP applications, both `Neurosurgeon` and MAUI correctly decide that local computation on the mobile device is optimal. However, MAUI makes incorrect offloading choices for more complicated scenarios (e.g., VGG, FACE, DIG and ASR). This is because MAUI relies on past invocation of a certain DNN layer type to predict the latency and data size of the future invocations of that layer type, leading to mispredictions. This control-centric prediction mechanism is not suitable for DNN layers because the latency and data size of layers of the same type can be drastically different within one DNN, and `Neurosurgeon`'s DNN analysis step and prediction model correctly captures this variation. For instance, in VGG, the input data size for the first and second convolution layers are significantly different: 0.57MB for `conv1.1`, and 12.25MB for `conv1.2`. For the mobile CPU and LTE, MAUI decides to offload the DNN before `conv1.2` due to its misprediction, uploading large amount of data and resulting in a 20.5× slowdown over the status quo approach. Meanwhile, `Neurosurgeon` successfully identifies that for this case it is

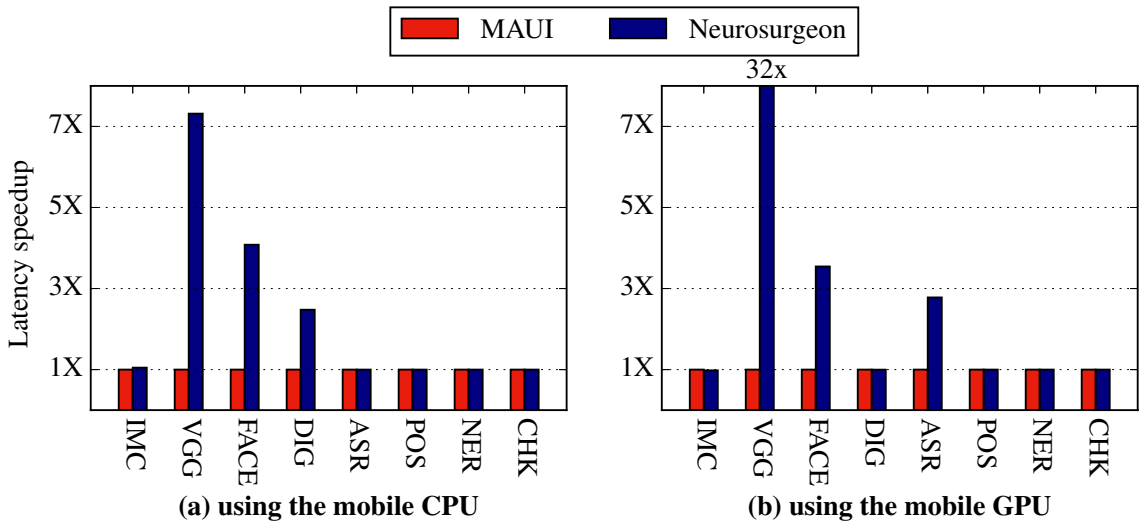



**Figure 3.11: Latency speedup achieved by `Neurosurgeon` vs. MAUI [35]. For MAUI, we assume the optimal programmer annotation that achieves minimal program state transfer. `Neurosurgeon` outperforms MAUI by up to 32× and 1.9× on average.**

best to execute the DNN entirely in the cloud, and thus achieves similar performance as the status quo and a 20.5× speedup over MAUI.

### 3.4.4 Network Variation

In this section, we evaluate `Neurosurgeon`'s resilience to real-world measured wireless network variations. In Figure 3.12, the top graph shows measured wireless bandwidth of T-Mobile LTE network over a period of time. The bottom graph shows the end-to-end latency of the status quo approach and `Neurosurgeon` executing AlexNet (IMC) on the mobile CPU platform. Annotated on the bottom graph is `Neurosurgeon`'s dynamic execution choice, categorized as either local, remote or partitioned. The status quo approach is highly susceptible to network variations and consequently the application suffers significant latency increases during the low bandwidth phase. Conversely, `Neurosurgeon` successfully mitigates the effects of large variations and provides consistent low latency by shifting partition choice to adjust the amount of data transfer based on the available bandwidth.

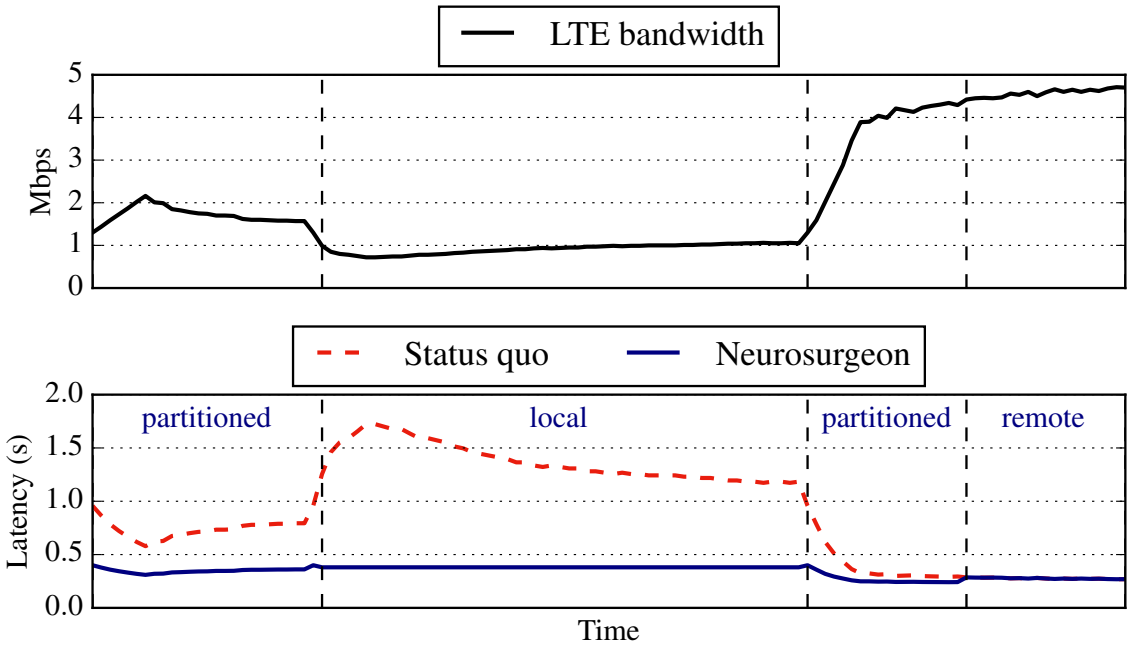



**Figure 3.12: The top graph shows bandwidth variance using a LTE network. The bottom graph shows the latency of AlexNet (IMC) of the status quo and `Neurosurgeon`. `Neurosurgeon`'s decisions are annotated on the bottom graph. `Neurosurgeon` provides consistent latency by adjusting its partitioned execution based on the available bandwidth.**

### 3.4.5 Server Load Variation

In this section, we evaluate how `Neurosurgeon` makes dynamic decision as the server load varies. Datacenters typically experience diurnal load patterns and high server utilization leads to increased service time for DNN queries. `Neurosurgeon` determines the best partition point based on the current server load level obtained by periodically pinging the server during idle period, and thus avoids long latency caused by high user demand and the resulting high load.

Figure 3.13 presents the end-to-end latency of AlexNet (IMC) achieved by the status quo approach and `Neurosurgeon` as the server load increases. The mobile device is equipped with a CPU and transfers data via Wi-Fi. As shown in the figure, the status quo approach does not dynamically adapt to varying server load and thus suffers from significant performance degradation when the server load is high. The

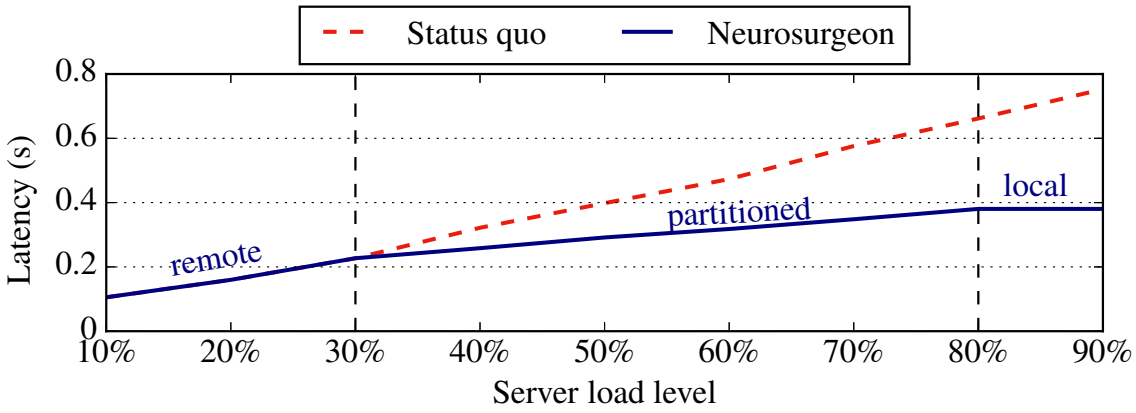



**Figure 3.13: `Neurosurgeon` adjusts its partitioned execution as the result of varying datacenter load.**

end-to-end latency of the status quo approach increases from 105ms to 753ms as the server approaches its peak load level. On the other hand, by taking server load into consideration, `Neurosurgeon` dynamically adapts the partition point. In Figure 3.13, two vertical dashed lines represent the points where `Neurosurgeon` changes its selection: from complete cloud execution at low load, to partitioning the DNN between mobile and cloud at medium load, and eventually completely onloading to mobile at peak load. Regardless of the server load, `Neurosurgeon` keeps the end-to-end latency of executing image classification below 380ms. By considering server load and its impact on the server performance, `Neurosurgeon` consistently delivers the best latency regardless of the variation in server load.

**Table 3.6: Comparing `Neurosurgeon` to popular computation offloading/partition frameworks**

| | MAUI [35] | Comet [38] | Odessa [70] | CloneCloud [28] | `Neurosurgeon` |
|---|---|---|---|---|---|
| **No need to transfer program state** | | | ✓ | | ✓ |
| **Data-centric compute partitioning** | | | | | ✓ |
| **Low/no runtime overhead** | ✓ | | ✓ | ✓ | ✓ |
| **Requires no application-specific profiling** | | ✓ | | | ✓ |
| **No programmer annotation needed** | | ✓ | ✓ | ✓ | ✓ |
| **Server load sensitive** | | | ✓ | | ✓ |

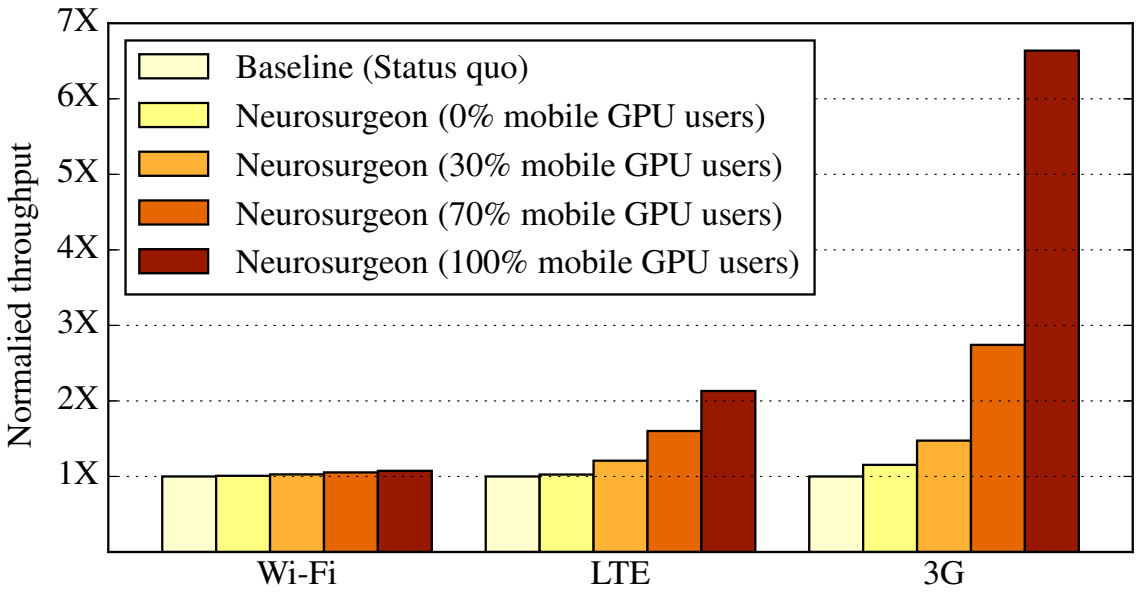



**Figure 3.14: Datacenter throughput improvement achieved by `Neurosurgeon` over the status quo approach. Higher throughput improvement is achieved by `Neurosurgeon` for cellular networks (LTE and 3G) and as more mobile devices are equipped with GPUs.**

### 3.4.6 Datacenter Throughput Improvement

`Neurosurgeon` onloads part or all of the computation from the cloud to mobile devices to improve end-to-end latency and reduce mobile energy consumption. This new compute paradigm reduces the computation required on the datacenter, leading to shorter query service time and higher query throughput. In this section, we evaluate `Neurosurgeon`'s effectiveness in this aspect. We use BigHouse [62] to compare the achieved datacenter throughput between status quo and `Neurosurgeon`. The incoming DNN queries are composed evenly of the 8 DNNs in the benchmark suite. We use the measured mean service time of DNN queries combined with Google web search query distribution for the query inter-arrival rate.

Figure 3.14 shows the datacenter throughput improvement normalized to the baseline status quo approach of executing the entire computation on the server. Each cluster presents results for a given wireless network type. Within each cluster, the first bar represents the status quo cloud-only approach, while the other four bars represent `Neurosurgeon` with different compositions of the mobile hardware. For example,

"30% Mobile GPU users" indicates 30% of the incoming requests are from mobile devices equipped with a GPU while the remaining 70% are from devices equipped only with a CPU.

When the mobile clients are connected to the server via fast Wi-Fi network, `Neurosurgeon` achieves on average 1.04× throughput improvement. As the wireless connection changes to LTE and 3G, the throughput improvement becomes more significant: 1.43× for LTE and 2.36× for 3G. `Neurosurgeon` adapts its partition choice and pushes larger portions of the DNN computation to the mobile devices as the wireless connection quality becomes less ideal. Therefore the average request query service time is reduced and a higher throughput is achieved in the datacenter. We also observe that as the percentage of mobile devices with GPU increases, `Neurosurgeon` increases the computation onloading from the cloud to mobile, leading to higher datacenter throughput improvement.

## 3.5 Compared to Prior Work

Previous research efforts focus on offloading computation from the mobile to cloud. In Table 3.6, we compare `Neurosurgeon` with the most relevant techniques on properties including whether there is heavy data transfer overhead, data-centric or control-centric partitioning, low run-time overhead, whether application-specific profiling is required, and whether programmer's annotation is needed.

In addition to these key differences, computation partition frameworks have to make predictions as to when to offload computation and the correctness of the prediction dictates the final performance improvements for the application. COMET [38] offloads a thread when its execution time exceeds a pre-defined threshold, ignoring any other information (amount of data to transfer, wireless network available, etc.). Odessa [70] makes computation partition decisions only considering the execution time and data requirements of part of the function, without taking the entire appli-

cation into consideration. CloneCloud [28] makes the same offloading decisions for all invocations of the same function. MAUI's [35] offloading decision mechanism is better in that it makes predictions for each function invocation separately and considers the entire application when choosing which function to offload. However, MAUI is not applicable for the computation partition performed by `Neurosurgeon` for a number of reasons: 1) MAUI requires a profiling step for each individual application, whereas predictions are required to perform DNN partitioning. `Neurosurgeon` makes decisions based on the DNN topology without any runtime profiling. 2) MAUI is control-centric, making decisions about regions of code (functions), whereas `Neurosurgeon` makes partition decisions based on the structure of the data topology that can differ even if the same code region (function) is executed. Layers of a given type (even if mapped to the same function) within one DNN can have significantly different compute and data characteristics. 3) `Neurosurgeon` transfers only the data that is being processed in contrast to transferring all program state. 4) MAUI requires the programmer to annotate their programs to identify which methods are "offload-able".

## 3.6 Summary

As an essential component of today's intelligent applications, Deep Neural Networks have been traditionally executed in the cloud. In this chapter, we examine the efficacy of this status quo approach of cloud-only processing and show that it is not always optimal to transfer the input data to the server and remotely execute the DNN. We investigate the compute and data characteristics of 8 DNN architectures spanning computer vision, speech, and natural language processing applications and show the trade-off of partitioning computation at different points within the neural network. With these insights, we develop `Neurosurgeon`, a system that can automatically partition DNN between the mobile device and cloud at the granularity of neural network layers. `Neurosurgeon` adapts to various DNN architectures, hardware

platforms, wireless connections, and server load levels, and chooses the partition point for best latency and best mobile energy consumption. Across 8 benchmarks, when compared to cloud-only processing, `Neurosurgeon` achieves on average 3.1× and up to 40.7× latency speedup, reduces mobile energy consumption by on average 59.5% and up to 94.7%, and improves datacenter throughput by on average 1.5× and up to 6.7×.

# CHAPTER IV

# Accelerating Deep Learning Based Natural Language Processing Applications

With the proliferation of deep learning approaches that provide state-of-the-art accuracy for a number of intelligent application domains, there has been a surge of recent work on accelerating deep learning computation. Meanwhile, the underlying deep learning algorithms themselves have begun to shift, becoming increasingly sophisticated and complex. This shift is particularly manifest in the domain of natural language processing (NLP), where there is a trend toward algorithms such as tree-structured long short-term memory neural networks (Tree-structured LSTMs) that have input-driven, dynamically-defined dependencies and structure, characteristics that cause these algorithms to map poorly to recently-proposed techniques for accelerating deep learning on GPUs.

In this chapter, we characterize, document, and taxonomize NLP applications to identify the algorithmic design elements and computational patterns that cause conventional GPU acceleration approaches to fail, providing a framework for guiding architects and system designers faced with supporting increasingly complex deep learning NLP applications in GPU equipped servers. Leveraging the insights gleaned from our framework, we design and implement *Fine-Grained Cross-Input Batching*, a novel fine-grain cross-input batching technique for providing GPU acceleration to

the dynamically-structured, input-dependent computations common to a number of state-of-the-art NLP applications. We evaluate our technique on real GPU system hardware on 3 difficult-to-accelerate NLP applications, improving throughput over a highly optimized CPU implementation by 7.6×, the GPU by 2.8×, and by 2.3× over the state-of-the-art GPU technique for deep learning.

## 4.1 NLP Applications Algorithmic Structure

In this section, we describe the set of natural language processing (NLP) applications investigated in this work, and their underlying algorithmic structures.

**Table 4.1: Application specifications**

| Application | Network | Input | Input Length | Description |
|---|---|---|---|---|
| LSTM [80] | LSTM | Movie Reviews [11] | 2 - 67 Words | Sentiment Analysis |
| NAMAS [73] | DNN | News Articles [5] | 1 - 20 Words | Text Summarization |
| CNN [50] | CNN | Movie Reviews [11] | 2 - 67 Words | Sentiment Analysis |

Previous works that characterize and accelerate [23, 25, 27, 36, 43, 58, 65, 69] deep learning workloads focus almost entirely on applications with fixed amount of neural network computation for each query. For example, image classification and face recognition applications feed a fixed size input image to a convolutional neural network for prediction result. Due to the intrinsically complicated nature of human natural language, the state-of-the-art NLP applications have begun to leverage more complicated algorithmic structures and computational patterns to better capture the semantic and syntactic structure of the natural language input. In the remainder of this section, we briefly describe the underlying algorithms used in our suite of applications.

**TreeLSTM.** Tree-structured Long-Short Term Memory Neural Network (TreeLSTM) is designed to capture input sentence's semantic structured in the form of a dynamically-formed binary parse tree, where the leaf nodes of the tree representing

words in the sentence and internal nods of the tree represent phrases [80]. The algorithm traverses through the tree and compute result for each node using the results of all of its child nodes as input.

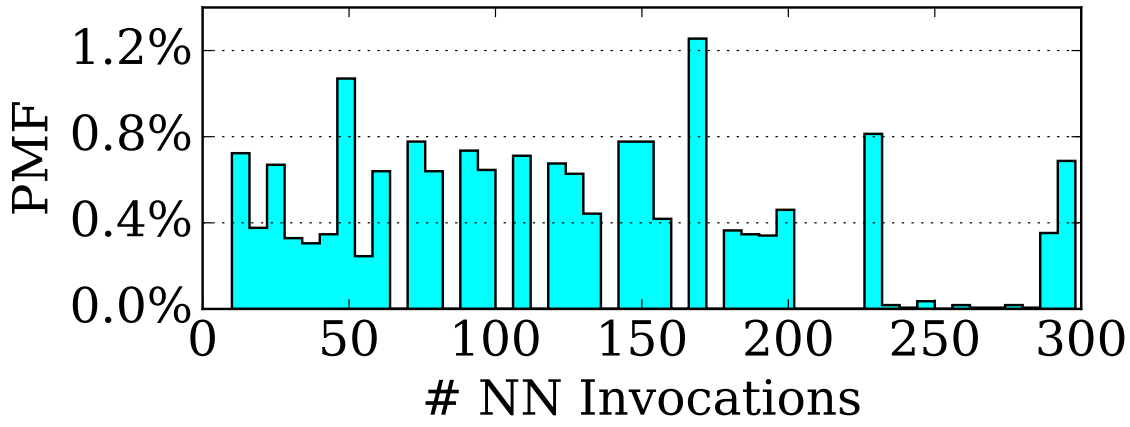
**NNLM.** Neural network language models (NNLM) are widely used in speech recognition [68] and machine translation systems [84]. This approach is designed to model the probability of a sequence of words in a language context. The summarization application NAMAS uses NNLMs to model the probability of a certain word appears in the output summary based on the previously-generated sequence of summary words.

**CNN.** As the de-facto neural network architecture used in the field of computer visions, Convolutional Neural Networks (CNN) have also been used to solve NLP problems [32]. For example, CNN has been applied to the problem of binary sentiment analysis [50].

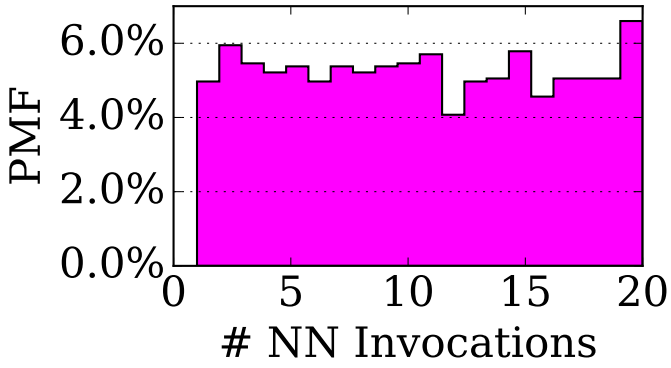
We summarize the suite of applications in Table 4.1 with the underlying algorithms, source of training data and range of input length of each application. For all applications, we use the implementation open-sourced by the papers' original authors.

## 4.2 Characterization

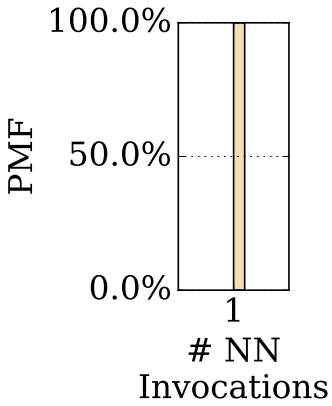
In this section, we characterize the 3 state-of-the-art NLP applications shown in Table 4.1. In two of these applications, the algorithm and the underlying computational patterns depend heavily on the nature of the input, which dynamically influences the structure and dependencies within the computation. Specifically, this dynamism is presents itself as three fundamental differences from applications that have been characterized and accelerated by prior work: (1) the NN computation is iterative with dependence across iterations and a variable number of iterations based on the specific query, (2) the total time spent in NN computation is smaller for NLP applications, and (3) NLP applications employ smaller NN kernels. Given these dif-




(a) LSTM




(b) NAMAS




(c) CNN

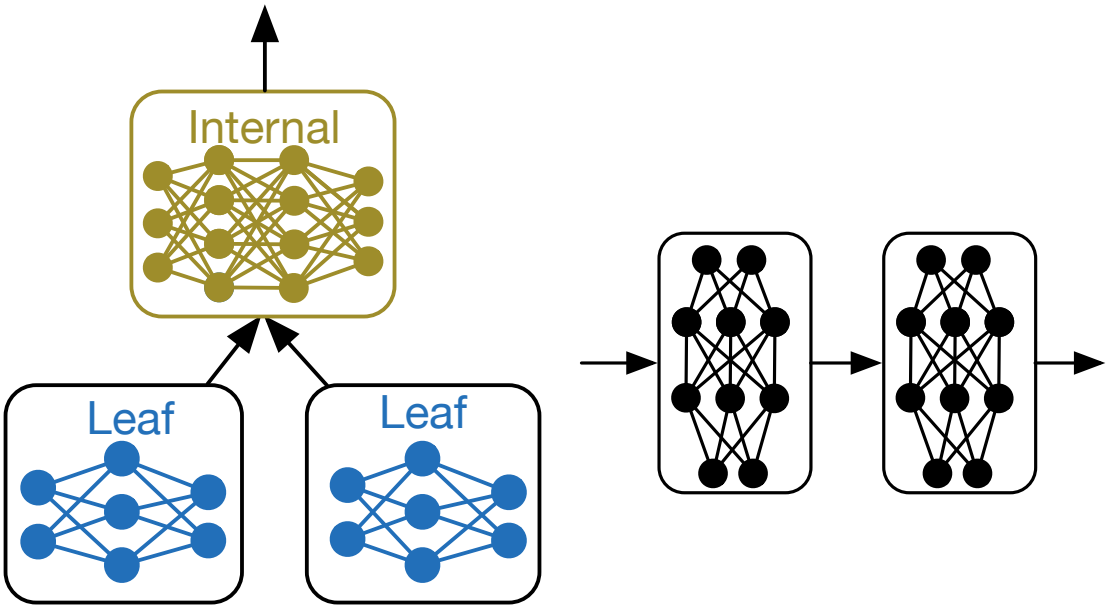
**Figure 4.1: NN invocations variability**

ferences, we find that the most recent neural network acceleration techniques are not suitable for these workloads.

### 4.2.1 Varying and Dependent NN invocations

Since state-of-the-art NLP applications process queries on a word-by-word basis, with each word depending on the last, their computational pattern is intrinsically iterative. When compared to applications that require a single DNN execution, these iterative DNN computations result in two key differences, varying NN invocations and dependent NN processing.

**Varying NN Invocations.** Multiple neural network inferences are invoked to




(a) Hierarchical Dependency (b) Linear Dependency

**Figure 4.2: NN invocations dependency**

process each NLP query. The number of NN invocations varies significantly from query to query. We show this in Figure 4.1, which presents the probability mass function (PMF) of the number of NN invocations for each application. From the figure, we find that LSTM and NAMAS exhibit high variance in the number of NN invocations. LSTM has this behavior, since it processes sentences by traversing a parse tree that represents the syntactic structure of the sentence, where the number of NN invocations depends on the number of nodes in the parse tree. Similarly, NAMAS generates output summaries on a word-by-word basis, where the number of NN invocations equals the number of words to be generated in the summary. In contrast, computer vision applications apply only one NN inference for each input image.

**Dependent NN Invocations.** NN-based NLP applications depend on part or all of the outputs from prior NN iterations for each query. As shown in Figure 4.2, these dependencies take two common forms, (a) hierarchical dependency and (b) linear dependency. We examine both of these dependencies in this work. NN invocations in

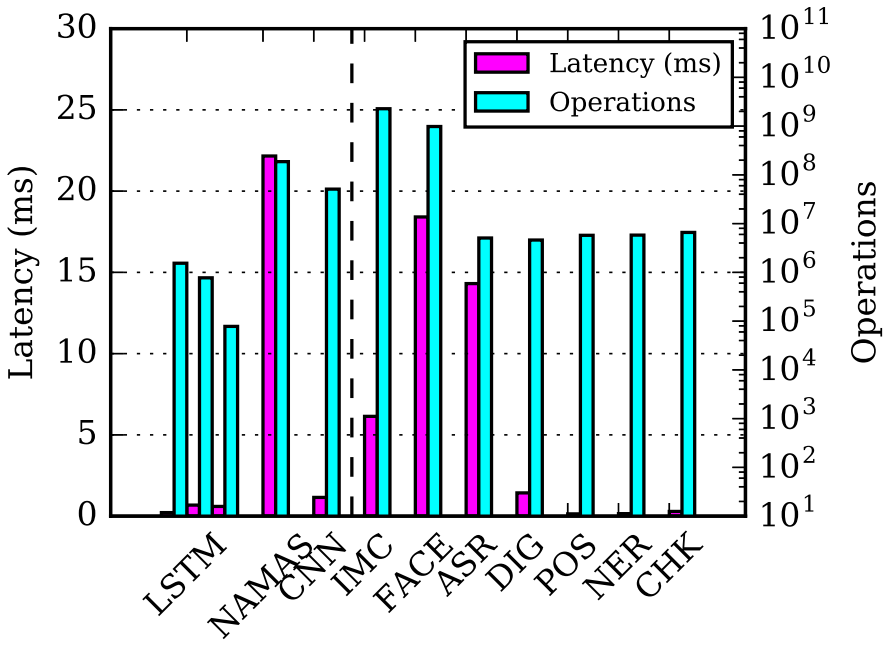



**Figure 4.3: Latency and FLOPS of NLP and traditional NNs on GPU**

LSTM follow a hierarchical dependency due to the syntactic parse tree being used to drive the processing of the input sentence. NAMAS has a linear dependency between NN invocations, since the generation of each word in the summary text is produced based on the previous one.

### 4.2.2 Few NN Kernel Computations

Intuitively, the iterative nature of NLP applications lends itself to smaller NN kernels, since a NN invocation processes a single word, compared to, for example, an entire image. We characterize this difference in Figure 4.3, which shows the number of floating-point operations per NN invocation and the corresponding GPU latency for a number of NN applications. These applications include the key NLP applications studied throughout this work (left) as well as those from Tonic Suite [43] (right).

From this figure, we find that NNs used in NLP applications require significantly fewer operations than traditional NNs. On average, NNs used in conventional NN

applications require 23× more operations than those in these NLP applications. In some cases, the difference is so substantial (e.g., 5K operations in LSTM, compared to over 1G operations in IMC or FACE) that it is unclear whether current acceleration techniques are amenable to these NLP workloads – the benefits of improving compute may be outweighed by communication overheads required for acceleration.

As a result of smaller compute requirements, latency of NLP NNs are smaller than traditional NNs, as shown in Figure 4.5. On average, NLP NNs take 3.6ms to execute on the GPU, among which LSTM NN latency is in the sub-millisecond range. On the other hand, traditional NNs takes 10.1ms to execute on the GPU on average and IMC,FACE and ASR cost more than 5ms.

### 4.2.3 Cycles Spent in NNs

Because NLP applications require several short-running NN invocations, each requiring significant preprocessing computation, we expect that NLP applications tend to spend a much larger fraction of time outside of the NN invocations. To investigate this difference, we partition the NLP applications into NN computations and non-NN computations. The fraction of time spent in each of these partitions, when running the non-NN work on the CPU and the NN work on the GPU, is provided in Figure 4.4.

As expected, the NLP applications spend a significant portion of the time outside of the NN computation, contrary to traditional NN applications [43]. To explain this difference, we provide the key sources of non-NN computation for each of the NLP applications. In LSTM, the application parses the sentence to generate a parse tree at the beginning of the processing and traverses the parse tree between invoking NN inferences at each tree node. In NAMAS, a beam search is conducted after each candidate summary words is generated to keep track of the search space. In CNN, the word embedding for each word in the input sentence is looked up in a large table.

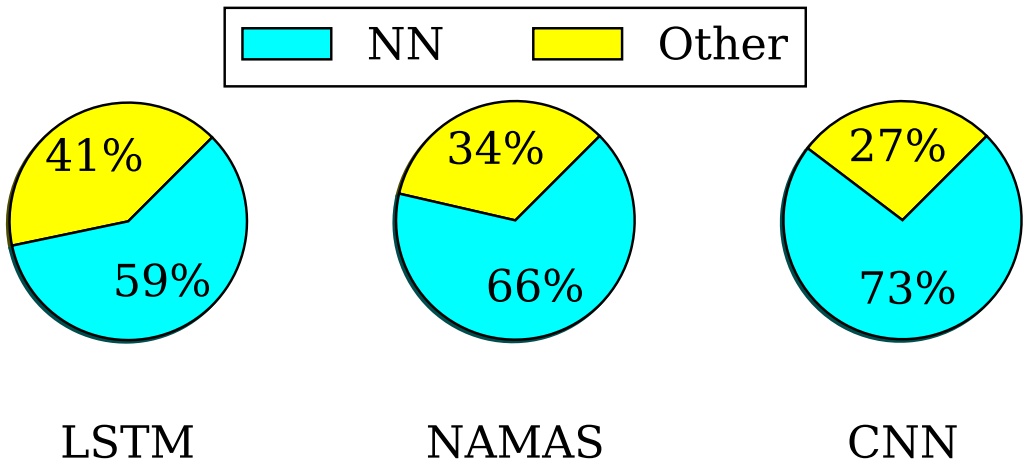



**Figure 4.4: Cycles breakdown of each applications. NN executes on the GPU and rest of the applications executes on the CPU**

On average, the suite of NLP applications spend about 34% of the execution time doing non GPU-amenable execution.

### 4.2.4 Limitations of Prior Work

Prior work has proposed a technique to accelerate deep learning applications on GPUs. Specifically, DjiNN and Tonic [43] observes low GPU occupancy when executing NNs. To increase GPU utilization, `DjiNN` batches multiple NN inputs together and executes them in parallel. The paper shows that this batching method provides higher throughput gains for NNs with lower occupancy. To evaluate the approach in `DjiNN` on accelerating NLP applications, we measure the GPU occupancy of each NLP application, and apply `DjiNN`'s batching approach to the NLP applications. The relation between throughput gain and GPU occupancy is shown in Figure 4.5.

**Traditional NNs.** Similar to the analysis presented in [43] , we observe that `DjiNN` applications with lower occupancy tend to scale better with increasing batch size. POS, NER and CHK have occupancy lower than 20% prior to batching and they exhibit the highest throughput gain among the 7 `DjiNN`'s applications studied. On

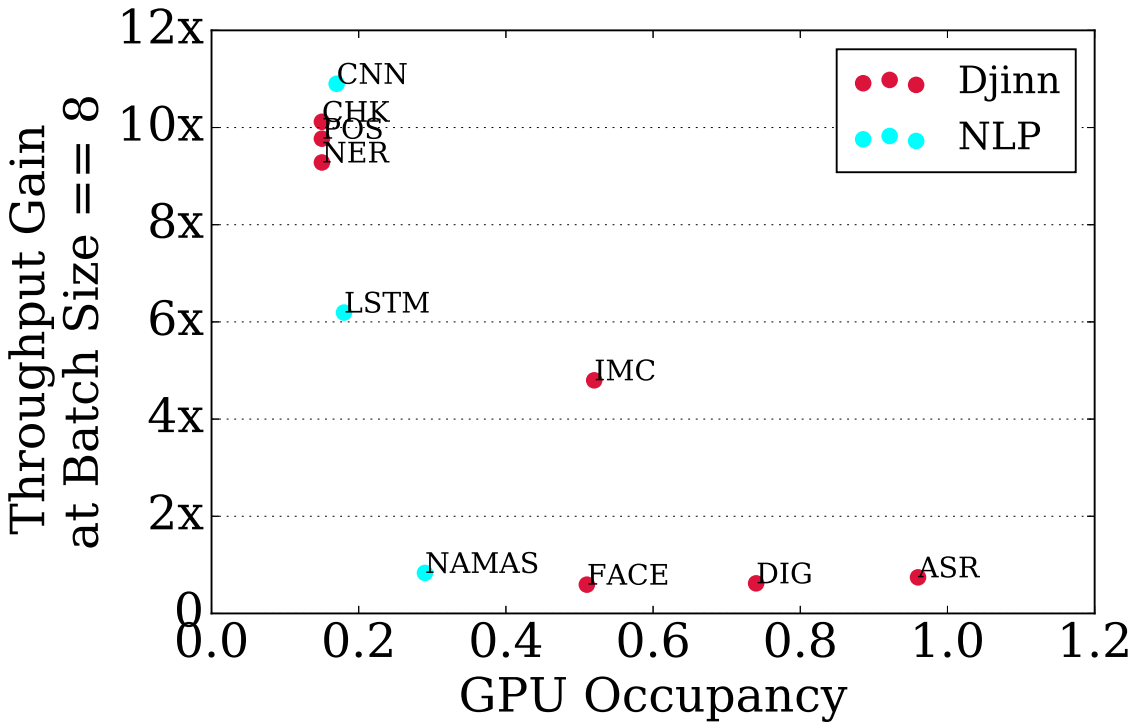



**Figure 4.5: Occupancy and throughput gain of applying DjiNN's batching technique**

the other end of the spectrum, ASR and DIG have an occupancy of 0.96 and 0.74 respectively and they benefit significantly less from batching.

**NLP Applications.** The red points in Figure 4.5 represents the throughput gain and occupancy for the suite of NLP applications. These 3 NLP applications share similar occupancy with POS, NER and CHK (between 15% and 30%). However, the throughput gains of NLP applications are not directly correlated to the occupancy. Specifically, LSTM and NAMAS experience limited throughput gain from `DjiNN`'s batching. This shows that GPU occupancy is not sufficient to derive throughput gain from batching and DjiNN's batching technique is not suitable for accelerating these NLP applications.

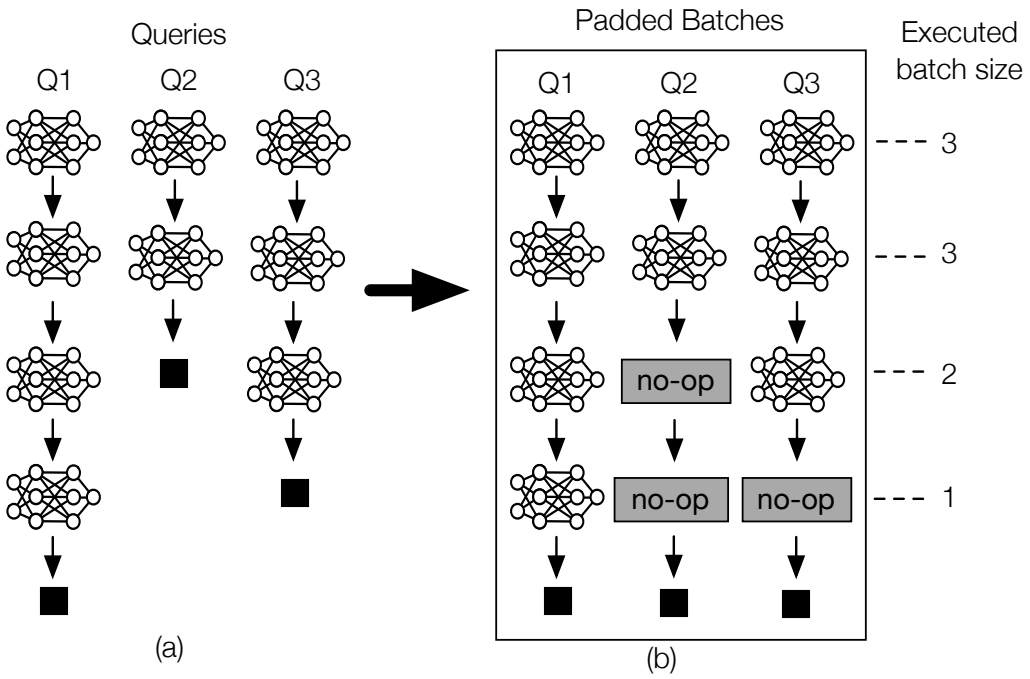



**Figure 4.6: Padding to Batch**

## 4.3 Applicability of State-of-the-art

As observed in the previous section, the current state-of-the-art DNN high throughput system proposes using GPU occupancy as the metric to evaluate throughput improvement for deep learning based applications. However, this metric is not sufficient to evaluate potential benefit from batching because simply batching does not directly translate to throughput gains. In this section, we investigate in detail the reason behind the unexpected low throughput gain from applying the batching proposed in prior work [43], quantify the inefficiencies of this state-of-the-art technique, and propose a new metric to consider alongside GPU occupancy to inform the design of the system.

### 4.3.1 Padding to Batch

The batching technique proposed in prior work makes assumption that all queries have the same neural network topology (computation) meaning each query executes the same neural network architecture. For the applications in DjiNN, each query has only one invocation of a single type of NN. Conversely, the NLP applications studied in this work have varying number of NN calls. LSTM and NAMAS queries vary in terms of the number of NN invocations, as shown in Figure 4.1. LSTM queries require invocations of different types of NN computation depending on the position in the traversal of the tree (as illustrated in Figure 4.2).

In order to apply prior work's batching technique, queries of varying length are padded to the longest query in that batch. This is a required step in order to be able to execute the batch at runtime. Figure 4.6 illustrates DjiNN padding where a batch is formed and all queries are padded to the same length (in this case to 3 NN boxes). The dependencies between the NNs (illustrated by arrows between the boxes) forbids batching queries in both dimensions (within a query and across queries). The result is wasted computation (gray boxes) which for this example represents 33% of the computation that could be spent doing meaningful work.

### 4.3.2 Quantifying Wasted Computation

Batch padding achieves higher GPU utilization that is misleading because only part of the work on the GPU contributes to queries making progress in their execution. Figure 4.7 shows the amount of computation wasted as batch sizes increases. We use a trace of randomly generated queries that have input lengths in the query length ranges described in Table 4.1. As soon as there are enough queries to form a batch, all queries will be padded to the longest batch. As batch size increases, the range of query lengths within a single batch increases meaning more queries must be padded. At batch size

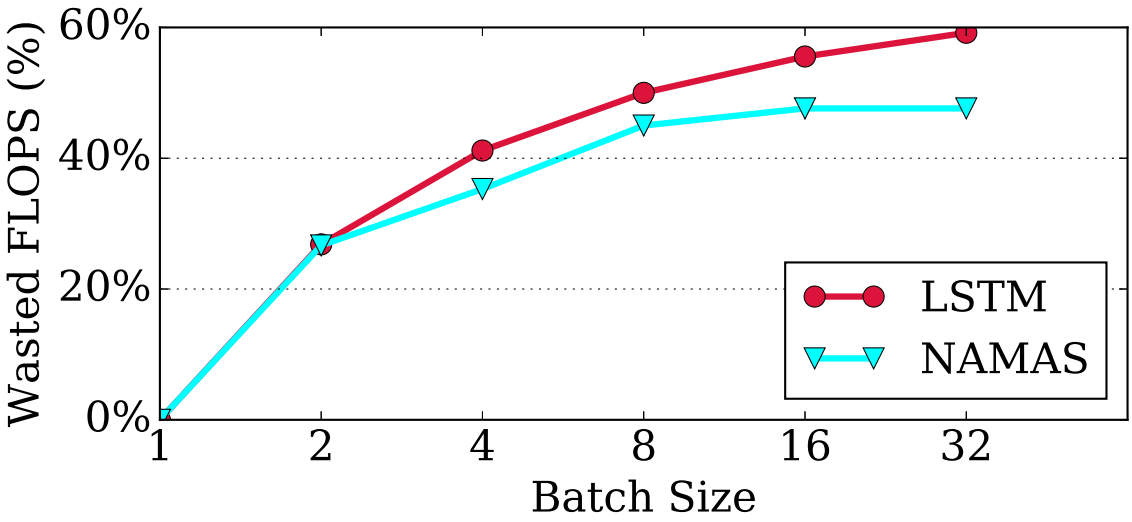



**Figure 4.7: Percentage of FLOPS wasted from padding**

of 32, up to 60% of the computation is unnecessary computation, significantly wasting computation on the GPU. This explains the low throughput gains from padding for the NLP applications studied in this work in Figure 4.5.

### 4.3.3 Revisiting NN Application Taxonomy

After showing the ineffectiveness of state-of-the-art batching techniques for the suite of NLP applications studied in this work, we propose an improved taxonomy of NN applications to better inform system architects when making design choices. Alongside the occupancy from Figure 4.5, we use the NN computation variance as the metric to be considered when evaluating the effectiveness of batching for NN applications. As we show in this section, the varying query-length translates into different amount of NN computation within a single query.

Figure 4.8 shows the characterization of the 7 traditional NN applications and 4 NLP applications studied here. As shown in Figure 4.5,prior work's batching technique benefits applications with 1) low GPU occupancy and 2) no variation in their NN computation. We use the coefficient of variation of the number of NN invoca-

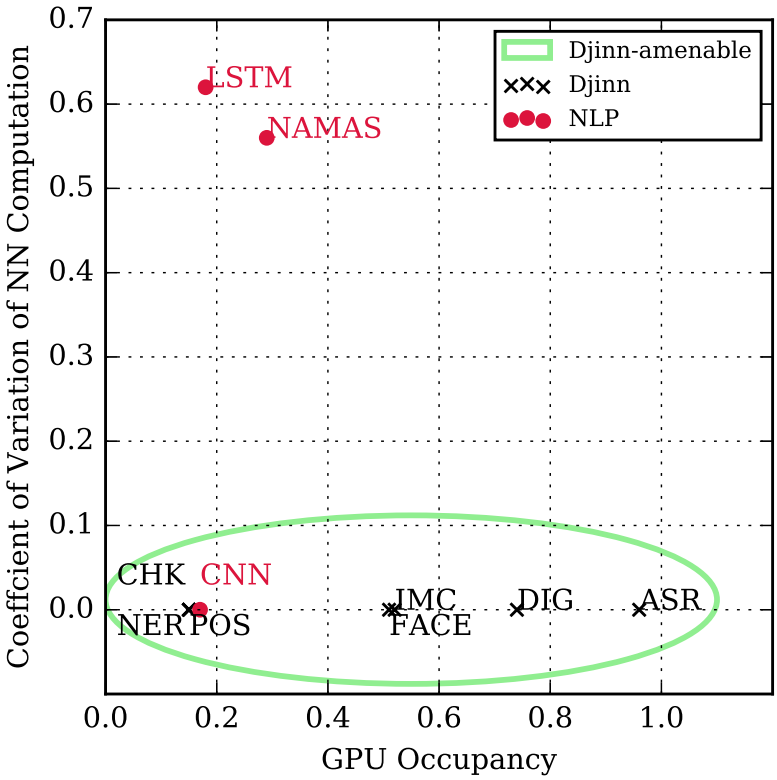



**Figure 4.8: Taxonomy of NN applications**

tions per query to represent the NN computation variation of the NLP application studied in this work. This metric separates LSTM and NAMAS from the rest of the applications, indicating that they are not amenable to DjiNN acceleration.

This in depth characterization shows severe flaws in current systems that strive to achieve high throughput for deep learning. This suggests a new system is needed to address the challenges exposed from this suite of NLP applications.

## 4.4 Designing a High Throughput Engine for NLP

In this section, we present the design of our system to address limitations of current systems. Figure 4.9 represents the architecture of the fully built out end-to-end system.

### 4.4.1 Requirements

NLP applications have three core characteristics that our system aims to address: 1) they have dependent DNN calls rendering intra-query batching impossible, 2) they are input length variable handicapping current techniques used to achieve high throughput, and 3) they have iterative and small NN computation making batching even more critical. We design a runtime system that addresses these limitations while providing additional benefits for applications that require traditional batching techniques. We target the following objectives:

1. **Dependency and Input Length Agnostic Batching** The system must be able to form optimal batch sizes if NN computation is available (irrespective of any dependencies between DNN calls). The system needs to be able to form a batch across queries regardless of their variable input length.

2. **High Throughput** The system must deliver and sustain high throughput given the multiple stages of such a large system needed to build an end-to-end system.

3. **Scalable** A scalable batching system must be able to scale with the amount of resources available and service incoming requests at high load. The system must have a flexible design to allow CPU and GPU resource tuning to optimally allocate resources.

### 4.4.2 System Design

#### 4.4.2.1 Fine-Grained Cross-Input Batching

The nature of text based applications make them difficult to batch because sentences are of variable length and have structure (ordering) to be grammatically cor-

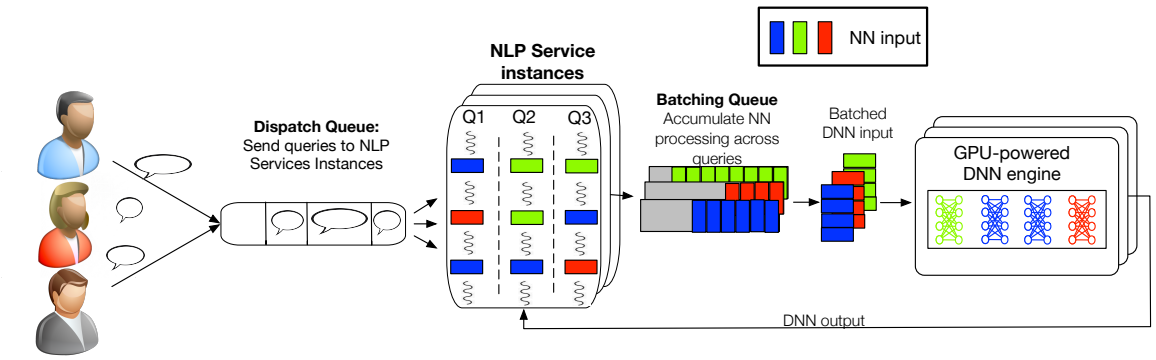



**Figure 4.9: System Design Overview**

rect. We introduce *Fine-Grained Cross-Input Batching (FGCIB)*, fine-grained cross-input batching to allow batching NN batching across multiple queries.

Our system collects NN computation across multiple queries to form a batch of NN computation that has independent NNs within a single batch. As shown in Figure 4.9, an NLP instance can have multiple queries in flight each requiring NN computation along the execution of a single query. The CPU worker will execute the CPU portion of the query until it meets NN computation (colored box in the diagram) at which point it will place the NN computation in the Batching Queue, save the progress of that query, and suspend its execution. The CPU worker is now free to service new incoming queries and repeat the process. At a given batch size, the DNN engine will pull the queries from the queue, batch the input, execute the batched NN computation, and make a callback to the NLP service signaling the queries can resume their execution.

Prior work proposes building a system that sends NN work over the network to dedicated NN processing re- sources. Given the characteristics of the NN applications and their large communication to compute ratio, this is not a feasible design. Our system addresses this by integrating the NN batching engine directly into the application as a black-box drop-in library that provides a common implementation across the applications studied.

#### 4.4.2.2 Application Pipelining

To study these applications in a production environment, we build out the entire system that accepts queries over the network from a load generator, Treadmill [86] maintained and deployed at Facebook [6]. The system is composed of 4 stages: 1) a front-end dispatch queue that round-robins queries to the NLP instances, 2) the NLP instances each servicing queries, 3) the batching queue for each instance accumulating NN computation with a configurable timeout mechanism, and 4) the Batching Engine executing the NN computation on the GPU. These are effectively pipeline stages where the system throughput is bounded by the stage with the lowest throughput. Our system is fully asynchronous allowing threads to suspend and resume execution as queries progress through the system.

#### 4.4.2.3 Flexible Resource Allocation

Given the large variance in the breakdown of NN vs non-NN computation of the applications studied, the system needs to provide flexibility in where computational resources can be applied. The number of threads dedicated to serving incoming queries are referred to as CPU Workers. Within a single instance, multiple workers can be processing queries in parallel pushing NN computation to a single unified queue. This reduces the time it takes to form a batch of queries as there are now multiple workers pushing compute to a single queue. The system can also be scaled up with the number of instances of the entire system providing a tunable parameter between instances and workers.

### 4.4.3 Configuration Tuning

Our system features a set of tunable parameters to ensure flexible adaption to NLP applications, which despite having a common set of computational patterns, exhibit differences in particular characteristics such as the size and structure of the

neural networks, and input sizes. The tunable parameters within the system include: Our system allows the tuning of three parameters for each application:

1. Batch size
2. Number of CPU workers to pair with each GPU processing instance
3. Total number of service instances

The combinations of these parameters amount to hundreds of different configurations for a large-scale system, making it tremendously time-consuming to exhaust every option in the design space in order to find the configuration that will most efficiently utilize the underlying hardware resource. Instead, the following approach is used to determine these parameters for a specific NLP application:

1. **FGCIB Batch Size Scaling** We first scale the batch size of the NN in the application and measure GPU occupancy and system throughput. Increased batch size creates larger problem size for the GPU, increases GPU occupancy, and improves throughput of the NN processing stage. Based on the resulting occupancy, we limit the possible choice of batch size within the batch sizes that provided the highest throughput gain at the highest occupancy, and eliminate the batch sizes that are either too small to provide significant throughput gain or too large that they provide diminishing returns.

2. **Balancing CPU and GPU resources** Increasing batch size allocates more GPU resource for the NN processing stage by better utilizing the GPU resources. The CPU is responsible of preprocessing each query and the computation between each NN invocation (for example traversing a tree). A balance of the resources allocated for the two stages is required to keep the individual throughputs comparable to minimize waiting time between the pipeline stages. We model the throughput of the CPU stage as it's using more CPU threads

and model the throughput of the GPU stage as the batch size increases. We limits the configurations to have a combination of (Batch Size, number of CPU workers) that achieves similar throughput for the two stages. We eliminate any imbalanced configurations that may lead to one stage idling excessively.

3. **Service Instance Scaling** We now considers the scaling up the number of individual service instances to maximally utilize the underlying hardware resources. Scaling service instances involves allocating more CPU threads and having multiple GPU context executing on the GPU simultaneously. To further pruning the candidate configurations, we evaluate the overall GPU utilization of a single instance with configurations survived through the previous two filtering steps. We then identify the configurations with the highest throughput but the lowest GPU utilization at single instance - indicating the highest GPU resources potential to be harvested by multiple GPU contexts. We now have arrived at the best configuration. Next we scale the number of instances configured at said best configuration up to using all available CPU threads or all available GPU Streaming Multiprocessors.

Later in section 4.5.2 we show how to derive the best configuration for each NLP application in our suite using the configuration filtering framework and evaluate the performance of our system at such configuration.

## 4.5 Evaluation

We next evaluate FGCIB, documenting our observations and its efficacy in accelerating NLP applications with irregular computational structures.

### 4.5.1 Methodology

**Applications.** We evaluate our system across all three applications – NAMAS, Tree-structured LSTM and CNN. NAMAS and LSTM covers a space of applications that are difficult to accelerate using conventional approaches, while CNN represents an application that is suitable for acceleration with conventional approaches, showing the applicability of this system to traditional, fixed topology applications.

**Platform.** Our experimental setup uses a client-server architecture, where a load-generating client running an industry-grade load generator [86] sends queries to our server over the network. The server uses the FGCIB fabric described in section 4.4 to process the queries, returning responses back to the client over the network. Queries are sent following an exponentially distributed inter-arrival rate, as prior research shows such a distribution accurately models production query arrival times [61]. The queries are dispatched from the front-end dispatch queue to each service instance on the FGCIB Batching server for processing. The platform used for the batching server is a dual-socket Intel Xeon CPU E5-2630v3 running at 2.40GHz with 8-cores, 2-way HyperThreading and an NVIDIA Titan X GPU. One socket of the machine is dedicated to running the parser that is used by the Tree-LSTM to generate the tree before the LSTM is executed and one socket for the applications.

**Baselines.** We compare against three baselines - CPU, GPU and `DjiNN`. For all three applications we use highly optimized libraries to process the queries. NAMAS and the Tree-LSTM are written in Torch [30], a highly optimized deep learning library maintained by Facebook. The CNN uses Theano [19] a graph processing library. The DNN processing on the CPU is linked to MKL and the GPU linked to the libraries' respective highly optimized versions.

### 4.5.2 Performance Analysis

Figure 4.10 shows the throughput performance of single service instance of CPU baseline, GPU baseline, and our system. Figure 4.10 also shows the GPU occupancy of our system as batch size increases. The throughput of our system generally increases as we scale the batch size, overtaking the CPU and GPU baselines.

For NAMAS, throughput peaks at batch size of 4 and decreases beyond that where the system already achieves more than 60% of GPU occupancy at batch size of 4. There are two explanations for this: 1) the GPU's occupancy is near maximum so batching beyond 4 does not provide additional gains, and 2) the CPU is now the bottleneck since there is a substantial CPU piece required to process each NN call.

LSTM and CNN have relatively small NN kernels that have low occupancy. Batching provides significant benefits for both of these applications and we limit the batch size to 32 across all applications in our explorations as larger sizes are not practical. Interestingly, the Tree-LSTM GPU has lower throughput at small batch sizes but starts to see throughput gains beating the CPU at a batch size of 4. This is because the overhead of launching to the GPU at small batch sizes outweighs the benefits and that overhead is amortized at a larger batch size.

In some cases the CPU is a competitive baseline but the GPU is able to provide higher throughput at larger batch sizes by making use of more of its resources. The CNN has more layers (compared to the other applications) and is able to efficiently utilize the GPU significantly, outperforming the CPU and GPU baselines at larger batch sizes.

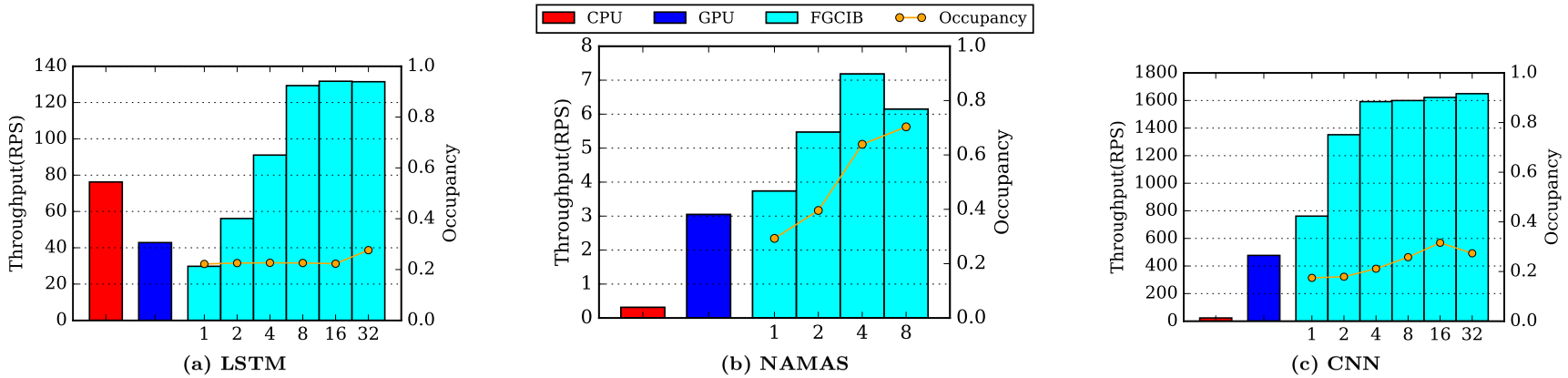
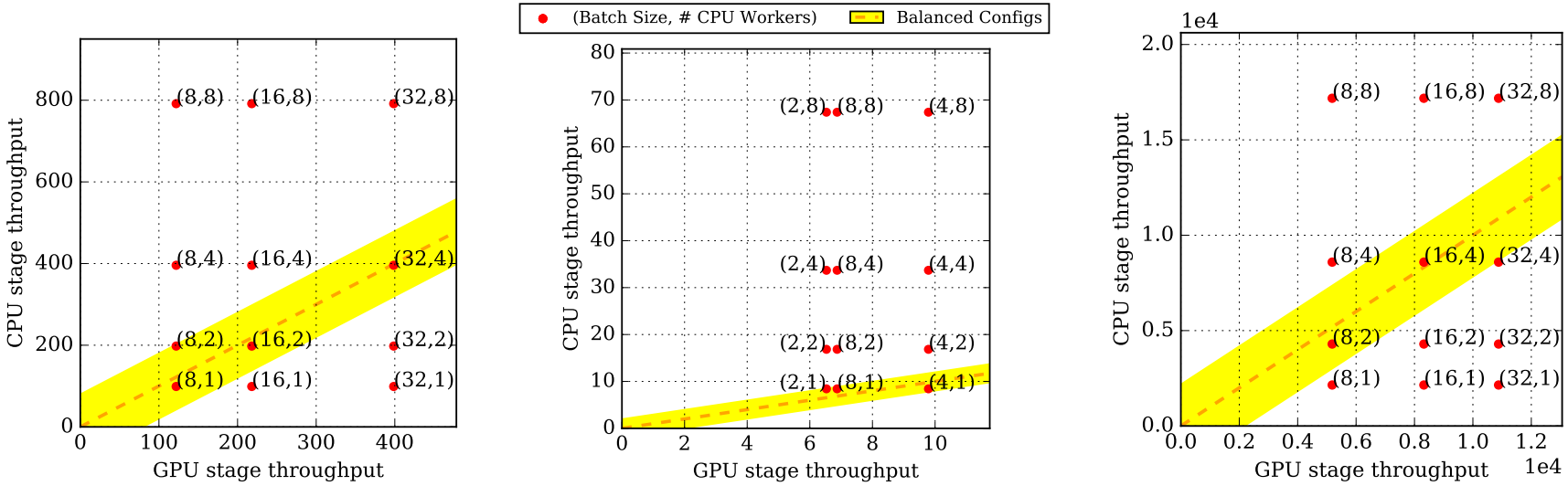



**Figure 4.10: End-to-end throughput and GPU occupancy of FGCIB with varying batch size**

### 4.5.3 Balancing CPU and GPU Resources

We now examine allocating CPU/GPU resources in the FGCIB system. One of the key problems in designing a system that makes the most of both the CPU and GPU is in balancing the resources on each side based on the amount of work required for each hardware. If either resource has a disproportionate amount of work to do over the other, that resource will be left idle and performance will be left on the table. The computation of a NLP query can be broken down into NN and non-NN work. In the FGCIB system, the non-NN portion of the computation is executed on the CPU, with each CPU thread handle one query, and the GPU handles the NN portion by executing batches of neural network input across multiple queries. In order to find the optimal balance between CPU and GPU resources, we identify configuration pairs of (number of CPU workers, batch size) that achieve similar service rate of the CPU stage and the GPU stage.

Figure 4.11 shows for each application, the throughput of the CPU stage w.r.t. the throughput of the GPU stage of different configuration pair. We model this by measuring latency of the unique CPU computation and NN computation on GPU with different batch sizes, which is a much smaller set of experiments than evaluating the end-to-end system with all the possible configurations. The line represents the optimal balance between CPU and GPU resources, where the CPU stage and GPU stage have similar service rate in their respective workload. Across all 3 applications, there are a set of candidate configurations (Batch Size, CPU Worker) that falls close to the desired balanced configuration. We extract these as viable candidates, pruning 70% of the configurations, to further drive our investigation into the best configuration for each application.

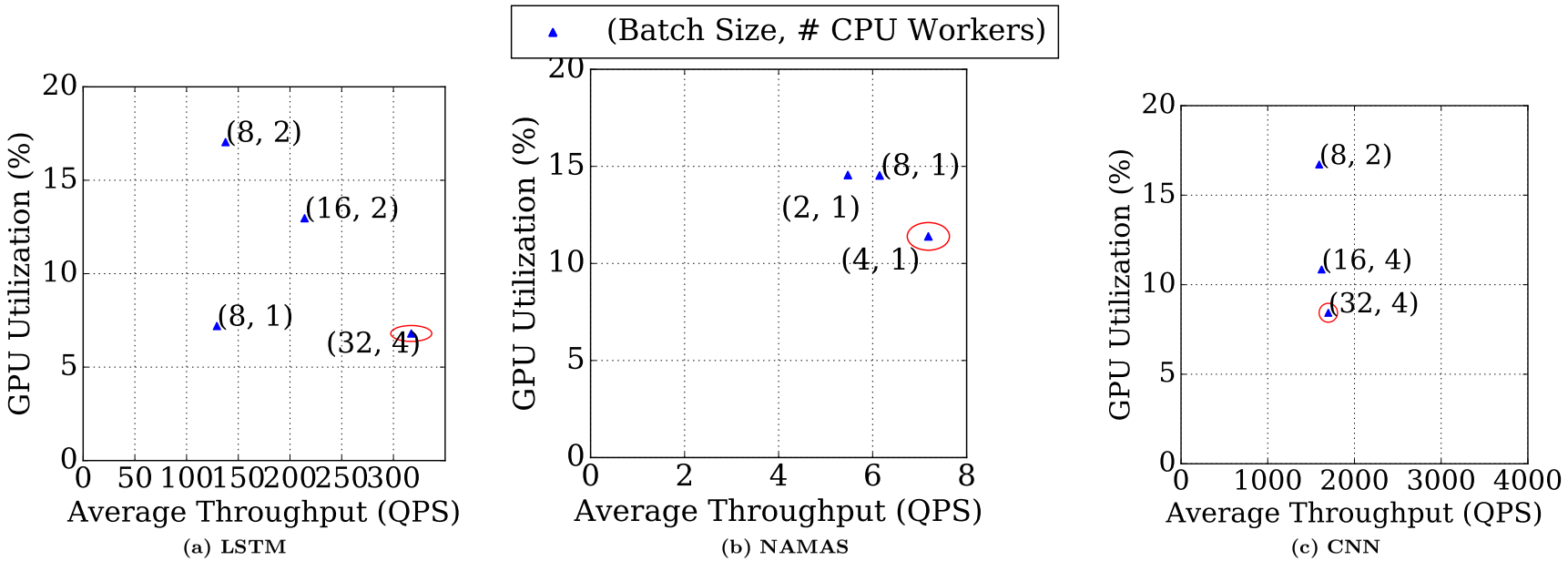



(a) **LSTM** (b) **NAMAS** (c) **CNN**

**Figure 4.11: CPU vs. GPU work balance across different numbers of workers and batch size**

**Scaling up instances** – We now scale up the number of independent service instances to fully utilize the GPU. Among the balanced configurations, our methodology selects the configuration which achieves the highest throughput with the lowest GPU utilization, to leave room for scaling up number of instances. Figure 4.12 shows the GPU utilization (occupancy) and achieved throughput of a single instance configured as the candidate configurations identified by the analysis above. The configuration with the lowest ratio between GPU utilization and achieved throughput is selected. For LSTM, this configuration is batch size of 32 with 4 workers; for NAMAS, batch size of 4 with 1 worker, and for CNN, batch size of 16 with 4 workers. We then scale up instances, each of which is configured with the selected configuration, until we fully utilize the GPU or occupy all available CPU threads.

**Key Insights** 1) There is a sweet spot in allocating CPU and GPU resources for systems that utilizes both hardware to process a single query. The most balanced configuration should achieve similar CPU stage throughput and GPU stage throughput to ensure most efficient resource utilization. For FGCIB, our configuration tuning algorithm selects the best configuration pair of CPU workers and NN batch size. 2) This set of balanced configurations varies across applications, underscoring the importance of careful per-application tuning according to the framework described in section 4.4.3. 3) To reduce the search space, we first select the configuration for an individual instance that has the lowest GPU utilization to throughput ratio and then scale up the number of instances following this configuration to fully utilize the GPU.




(a) LSTM (b) NAMAS (c) CNN

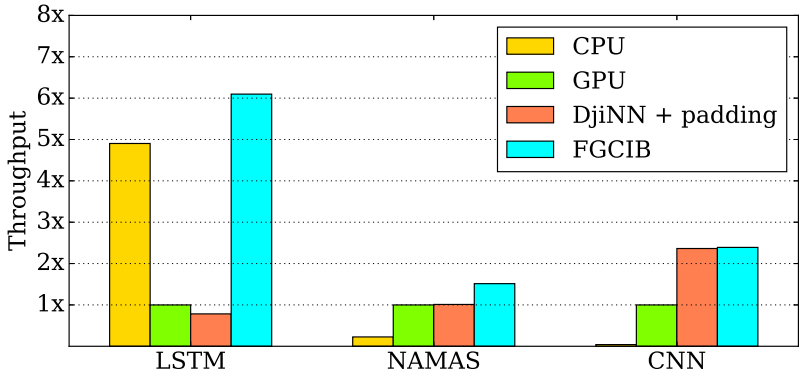
**Figure 4.12: GPU Utilization and Throughput prior to instance scaling**

### 4.5.4 Query Throughput and Latency

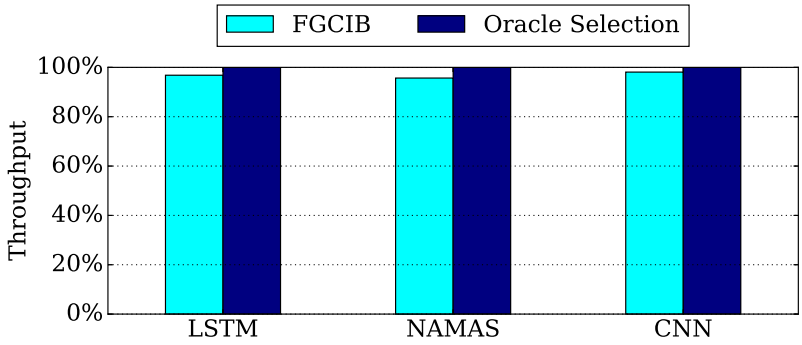
We now evaluate the throughput and latency performance of FGCIB. We compare against a CPU baseline and to DjiNN [43], the state-of-the-art prior work in batching deep learning queries to improve throughput on GPUs. This comparison covers FGCIB along with 2 versions of DjiNN, as described in section 5.1 – one version that implements the DjiNN technique as described in the paper [43], and a second technique that adds support for padding input queries to produce fixed-size queries to enable effective batching over the baseline DjiNN system.

**Throughput** – We first evaluate the throughput improvement achieved by FGCIB compared to the baselines. For FGCIB, we configure the system follow the final configuration derived in Section 4.5.3. For the CPU baseline and DjiNN baselines, we scale up the number of NLP application instances on the CPU and GPU, respectively, to the maximum number of instances.

Figure 4.13a shows the throughput achieved by FGCIB and the baselines. LSTM and NAMAS achieve throughput gains from the technique with NAMAS achieving 5× throughput improvement over the CPU. On average, FGCIB achieves 7.6× throughput improvement over the CPU. Our system achieves 2.8× higher throughput than GPU baseline (maximum number of instances sharing the GPU). When compared to DjiNN + padding (the state-of-the-art acceleration technique), our system on average achieves 2.3× higher throughput. Specifically, FGCIB achieves on average 4.5× higher throughput for LSTM and NAMAS while achieving slightly higher throughput for CNN. For CNN, this is because we see incremental gains from having an asynchronous, pipelined system. This demonstrates our system is significantly more effective than state-of-the-art at handling deep learning applications with dynamically defined computation and performs slightly better (no worse) compared to the state-of-the-art for traditional statically-defined deep learning applications.



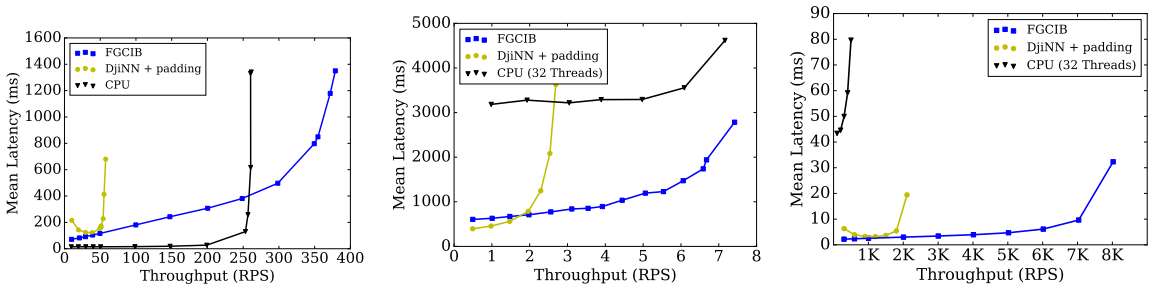

**(a) Throughput Improvement over GPU**




**(b) Throughput improvement compared to oracle configuration**

**Figure 4.13: Throughput Improvement**

**Query latency** – Figure 4.14 shows the mean query latency as a function of achieved throughput at different load for each application. We compare the latency and throughput of one FGCIB against a CPU baseline and DjiNN with padding. For FGCIB, we use the best configuration identified from the configuration tuning algorithm (Section 4.5.3). For DjiNN, we use the same number of threads as FGCIB for each application. For CPU baseline, we use the same number of threads as FGCIB for LSTM and use all threads available (32) for NAMAS and CNN. We allocate more than fair amount of threads to the CPU baseline of NAMAS and CNN because CPU performance is orders of magnitude worse than FGCIB for these two applications (Figure 4.13a) and difficult to visualize on the same graph. So for NAMAS and CNN, we use the generous CPU baseline configuration that occupies all CPU cores while compared to a single instance which only use some of the available cores. For this experiment, we employ a stochastic event-driven queueing simulation




**(a) LSTM** **(b) NAMAS** **(c) CNN**

**Figure 4.14: Mean service latency and throughput**

methodology borrowed from BigHouse [63] with service time distributions measured on real systems.

We first compare FGCIB against the state-of-the-art NN batching system, DjiNN. At high load level, FGCIB achieves significantly lower latency while achieving up to 3.5× higher throughput. At low load, FGCIB provides similar or lower latency. This performance difference stem from the ineffectiveness of DjiNN's batching method, padding queries to the longest length and wasting computation.

We then compare against the CPU baseline. For NAMAS and CNN, FGCIB achieves orders of magnitude lower latency compared against a CPU baseline using all available CPU cores. For LSTM, FGCIB achieves higher throughput with a lower latency at high load while has higher latency at low load. There are two reasons for this. First, when the query arrival rate is low, NN request is generated a lower rate and each NN batch takes longer to form. Secondly, LSTM features the smallest NN computation among the benchmark applications (Figure 4.3) and has the lowest GPU utilization (Figure 4.10a). It is inefficient to use GPU for LSTM computation of smaller batch size due to the data communication time and GPU kernel launch overhead. A single LSTM query is on average 40% faster to execute on CPU than on GPU. Our system mitigates this by creating large batch of NN queries to issue to the GPU at once to achieve higher GPU utilization and amortize the data communication/kernel launch overhead.

**Key Insights** 1) Traditional NN batching technique, which requires padding incoming padding queries to form batches, is not applicable to accelerating NLP applications. Padded queries waste resources and leads to long service time and low throughput. 2) A fine-grained cross input flexible batching scheme, like the one proposed with FGCIB, is crucial for efficiently forming large NN batches to utilize the GPU more efficiently. 3) Using FGCIB, applications achieve higher throughput with lower latency at higher load while achieve similar or better latency at lower load compared to state-of-the-art system.

### 4.5.5 Configuration Tuning Algorithm

We now evaluate the configuration tuning algorithm, by comparing the achieved throughput of the best configuration selected by the algorithm and an oracle selection which represents the highest possible throughput the system would achieve using a configuration identified after exhaustively experimenting with all possible configurations. Figure 4.13b shows the achieved throughput of the algorithm-selected configuration normalized to that of the oracle configuration. For all three applications, our methodology achieve throughput above 95.7% of an oracle configuration. Note that there is a large search space for the tunable parameters in the system and it is a very time-consuming to exhaustively experiment with most, if not all, of the possible configurations to derive at an oracle configuration.

## 4.6 Summary

Natural Language Processing (NLP) applications represent the next, relatively unexplored set of applications that system architects need to rethink their systems for. In this chapter, we thoroughly investigate and take a step in addressing new challenges that emerge from system design for NLP. The fundamental difference between traditional DNN based applications is in inherent nature of the inputs to the

NLP applications that require analysis of the individual words as well as their semantic position in the sentence. We identify three representative NLP applications that seemingly use the same algorithmic components (DNNs) but have drastically different computational characteristics. Through our in-depth characterization, we show that NLP applications have 3 main characteristics: 1) iterative and linear or tree-based dependent NN computation, 2) the computation per NN call is small, and 3) a significant fraction of the time is spent outside the NN.

These characteristics render current systems for high throughput DNN inference systems ineffective for this emerging class of NLP applications. We propose a novel batching technique, Fine-grained Cross-Input Batching (FGCIB) to address these characteristics as well as support traditional DNN type workloads. FGCIB allows batching at the level of NN computation across queries to eliminate the dependencies introduced within a query and allow queries of different length to be batched. We designed and implemented FGCIB and perform a real-system evaluation using an industry deployed load-generator, we achieve on average 7.6× throughput improvements over an optimized CPU baseline and 2.8× over the current state-of-the-art GPU acceleration system.

# CHAPTER V

# Data Collection for a Real-World Intelligent Application

Large, high quality training corpora are crucial in building effective machine learning models in many tasks required in building an intelligent application. The performance of the machine learning models, especially deep learning models, depend heavily on the quantity and quality of the training data. Developing real-world intelligent dialogue systems such as Apple Siri, Google Assistant and Amazon Alexa poses a significant challenge for data collection. In order to better understands the challenges, we experiment with building a dialogue system for a set of real-world use cases. We observe that the complexity of building real-world dialogue system is often substantially greater than those studied in the research community. For example, the dialogue system requires intent classification among 47 different intents, whereas most academic datasets for text classification only have a small number (i.e., 2–14) of classes [85]. The few datasets that have a large number of classes, such as RCV-1 [55], distribute intents across many distinct topics. The real-world application address the significantly more challenging problem of handling many intents within a single domain, specifically personal finance and wealth management, requiring the classifier to carefully distinguish between nuanced intent topics. Therefore, a large amount of high-quality training data tailored to our targeted problem is critical for creating the

best user experience.

Crowdsourcing offers a promising solution by massively parallelizing data collection efforts across a large pool of workers at relatively low cost. Because of the involvement of crowd workers, collecting high-quality data efficiently requires careful orchestration of crowdsourcing jobs, including their instructions and prompts, make effective crowdsourcing process largely an open research question.

In this chapter, we propose two novel metrics to evaluate dataset quality. Specifically, we introduce (1) `coverage`, quantifying how well a training set covers the expression space of a certain task, and (2) `diversity`, quantifying the heterogeneity of sentences in the training set. We verify the effectiveness of both metrics by correlating them with the model accuracy of two well-known algorithms, SVM [34] and FastText [20, 48]. We show that while `diversity` gives a sense of the variation in the data, `coverage` closely correlates with the model accuracy and serves as an effective metric for evaluating training data quality. We then leverage these metrics to evaluate multiple variants of crowdsourcing methods. Based on the insights we gained, we provide concrete recommendations on the best training data crowdsourcing practices.

## 5.1 Many-intent Classification

We focus on a specific aspect of dialogue systems: intent classification. This task takes a user utterance as input and classifies it into one of the predefined categories. Unlike general dialogue annotation schemes such as DAMSL [33], intent classification is generally domain-specific. Our system requires classification over 47 customer service related intents in the domain of personal finance and wealth management. These intents cover a large set of topics while some of the intents are very closely related and it requires the classifier to identify the nuanced differences between utterances. For example, user's queries to see a list of their banking transactions can often be very

similar to their queries to see a summary of historical spending, e.g., *"When did I spend money at Starbucks recently?"* vs. *"How much money did I spend at Starbucks recently?"*.

**Test Methodology** Our test data contains a combination of real user queries from a deployed system and additional cases manually constructed by developers. This combination allows us to effectively measure performance for current users, while also testing a broad range of ways to phrase queries. Our test set contains 3,988 sentences labelled with intents.

## 5.2 Training Data Quality Metrics

When we look to improve a model's performance, there are generally two approaches that we can take: improve the model and inference algorithm and/or improve the training data. There is currently no reliable way to help us identify whether the training data or the model structure is the current bottleneck. One solution is to train actual models using the training set and measure their accuracy with a predefined test set. However, if only a single algorithm is used, over time this evaluation may lead to a bias, as the training data is tuned to suit that specific algorithm. Using a suite of different algorithms avoids this issue, but can be very time consuming. We need an algorithm-independent way to evaluate the quality of training data and its effectiveness at solving the target task. In this section, we introduce two metrics to achieve this, `diversity` and `coverage`.

**Diversity** We use `diversity` to evaluate the heterogeneity of the training data. The idea behind `diversity` is that the more diverse the training data is, the less likely a downstream model will overfit to certain words or phrases and the better it will generalize to the testing set.

We first define a pairwise sentence distance measure. For a given pair of sentences, $a$ and $b$, we calculate the reverse of the mean Jaccard Index between the sentences' n-grams sets to represent the semantic distances between the two sentences:

$$D(a,b) = 1 - \frac{1}{N}\sum_{n=1}^{N}\frac{|n\text{-grams}_a \cap n\text{-grams}_b|}{|n\text{-grams}_a \cup n\text{-grams}_b|} \tag{5.1}$$

where $N$ is the maximum n-gram length. We use $N = 3$ in our experiments.

Our pairwise score is similar to the PINC score [22], except that we use the n-grams from the union of both sentences instead of just one sentence in the denominator of Equation 5.1. This is because the PINC score is used in paraphrasing tasks to measures how much a paraphrased sentence differ from the original sentence and specifically rewards n-grams that are unique to the paraphrased sentence. Our metric measures the semantic distance between two sentences and treat the unique n-grams in both sentences as equal contribution to the distance.

We define the `diversity` of a training set as the average distance between all sentence pairs that share the same intent. For a training set $X$, its `diversity` ($DIV(X)$) is:

$$DIV(X) = \frac{1}{|I|}\sum_{i=1}^{I}\frac{1}{|X_i|^2}\left[\sum_{a}^{X_i}\sum_{b}^{X_i} D(a,b)\right] \tag{5.2}$$

where $I$ is the set of intents and $X_i$ is the set of sentences with intent $i$ in the training set $X$.

**Coverage** We now introduce `coverage`, a new metric designed to model how well a training dataset covers the complete space of ways an intent can be expressed. We use our test set as an approximate representation of the expression space for our classification task. As described in § 5.1, our test set is constructed primarily with real user queries collected from the log of a deployed system and annotated by engineers.

To measure `coverage` of a training set given a test set, we first identify, for each

test sentence, the most similar training sentence with the same intent, according to the pairwise sentence distance measure $D(a, b)$ defined in Equation 5.1. We then derive `coverage` by averaging the shortest distances for all sentences in the test set. For a given test set, we would want the training set to have as high `coverage` as possible. Specifically, for a training set $X$ and a test set $Y$:

$$CVG(X, Y) = \frac{1}{|I|} \sum_{i=1}^{I} \frac{1}{|Y_i|} \sum_{b}^{Y_i} \max_{a}^{X_i} (1 - D(a, b)) \tag{5.3}$$

where $I$ is the set of intents and $X_i$ and $Y_i$ are the sets of utterances labeled with intent $i$ in the training ($X$) and test ($Y$) sets, respectively.

**Correlating Metrics with Model Accuracy** In order to evaluate the effectiveness of `diversity` and `coverage` at representing the training data quality, we collect training data via different methods and of varying sizes, train actual models, measure their accuracy and investigate the correlation between the metrics and the accuracy. We consider two well-known algorithms that have publicly available implementations: a linear SVM and FastText, a neural network-based algorithm.

**SVM** Support Vector Machines [34] are a widely used and effective approach for classification tasks. We use a linear model trained with the SVM objective as a simple baseline approach.

**FastText** We also consider a recently developed neural network approach [20, 48]. This model has three steps: (1) look up vectors for each n-gram in the sentence, (2) average the vectors, and (3) apply a linear classifier. The core advantage of this approach is parameter sharing, as the vector look-up step places the tokens in a dense vector space. This model consistently outperforms linear models on a range of tasks.

For all experiments we apply a consistent set of pre-processing steps designed to

| Type | Scenario | Paraphrasing |
|---|---|---|
| Generic | You want to learn about your spending history. | "Show me my spending history." |
| Specific | You want to learn about your spending history during a specific period of time. | "Show me my spending history in the last month. (Use different time periods in your answers)." |
| Generic | You want to ask about your income. | "What's my income?" |
| Specific | You want to ask about your income from a specific employer. | "How much money did I make from Company A? (Use different employers in your answers.)" |

**Table 5.1: Examples of generic and specific scenario description and paraphrasing prompts.**

reduce sparseness in the data: we lowercase the text, remove punctuation, replace each digit with a common symbol, expand contractions, and lemmatize (using NLTK for the last two).

## 5.3 Crowdsourcing Data Collection Methods

We consider two aspects of a crowdsourcing setup: the *template* style, and the *prompt*. The template defines the structure of the task, including its instructions and interface. Prompts are intent-specific descriptions or examples that define the scope of each task and guide workers to supply answers related to the target intent. We define a set of prompts for each intent and populate a template with each prompt to create a complete crowdsourcing job. We study two types of templates: scenario-driven (§ 5.3.1) and paraphrasing (§ 5.3.2), and two methods of generating prompts: manual generation (§ 5.3.1 and 5.3.2) and test set sampling (§ 5.4.3). A data collection method is the combination of a template and a prompt generation method. In this section, we describe each method and its variants.

**React to a Scenario**

Suppose you have a device that has a Siri-like app for your bank account that acts as a customer service agent and can handle questions about **your bank account balance.**

Given the original scenario described below that is related to your bank account, **supply 5 creative ways of asking the intelligent device to assist your situation**.

**"You want to ask about the balance of your bank account."**

**Figure 5.1: An example of scenario-driven task instructions. The template sets up a real-world situation and asks workers to provide a response as if they are in that situation. The prompt shown here is for collecting data for the intent 'balance'.**

### 5.3.1 Scenario-driven

The instructions for a scenario-driven job describe a real-world situation and ask the worker to provide a response as if they are in the situation. Figure 5.1 shows an example job for the intent of "asking about your bank account balance". Table 5.1 shows additional example prompts for generic and specific scenarios. Scenario-driven jobs simulate real world situations and encourage workers to create natural questions and requests resembling real user queries.

We consider two variations on the scenario-driven setup. Generic scenarios describe the situation in which the target intent applies, without specific constraints. For example, a generic scenario for the intent 'balance' is *"You want to know about your account balance"*. Specific scenarios refine the description by adding details. These are intended to encourage workers to write responses with more entities and constraints. These jobs also add specific information that the worker needs to include in their response. For example, a specific scenario for the intent 'balance' is *"You'd like to know the balance of one of your accounts. (Please specify the account you want to inquire about in your responses)"*.

For each intent, we use one generic scenario and three specific scenarios. To evaluate the different scenario types, we collected data with either generic scenarios

**Paraphrase Sentence**

Given the following sentence, **supply 5 creative ways of rephrasing the same sentence**.

Assume the original question is in regards to **your bank account balance**.

**"What is the balance of my bank account?"**

**Figure 5.2: An example of a paraphrasing task instructions.**

only, specific scenarios only, or a combination of both. The mixed setting contains equal contributions from all four scenarios (one generic and three specific). In our experiments, we keep the number of training samples per intent balanced across intents regardless of the number of total training examples.

### 5.3.2 Paraphrasing

Paraphrasing jobs provide an example sentence and ask workers to write several creative paraphrases of it. Figure 5.2 shows an example of job instructions for paraphrasing the sentence *"What is the balance of my bank account?"* To make sure we can directly compare the results of paraphrasing and scenario-driven jobs, we convert each scenario used in § 5.3.1 into a user question or command, which is provided as the sentence to paraphrase. As a result, there are two types of paraphrasing prompts: generic prompts and specific prompts. Table 5.1 shows example pairs of scenarios and paraphrasing prompts. Like in the scenario-driven case, we construct training sets with three different mixes of prompts, generic only, specific only and a combination of both.

## 5.4 Evaluation

In this section, we first verify that `diversity` and `coverage` provide insight regarding training data quality. We compare trends in these metrics with trends in

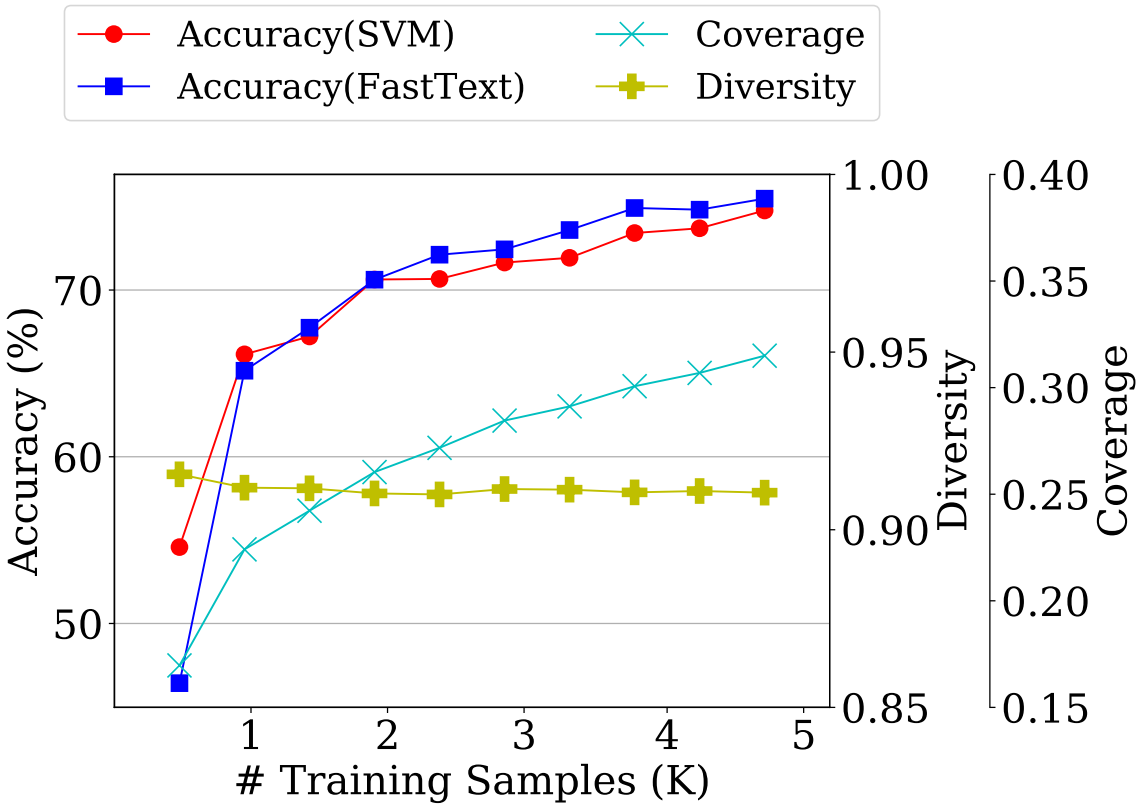



**Figure 5.3: Accuracy, `coverage` and `diversity` for scenario-driven jobs as the training data size increases. This data is collected using a mixture of generic and specific scenarios.**

model accuracy as the amount of training data is increased. We then evaluate the performance of the scenario-driven and paraphrase methods and their variants by comparing the quality of training data collected via these methods. Finally, we explore sampling paraphrasing examples from the test set and compare against manually generation by engineers.

### 5.4.1 Correlating Diversity and Coverage with Model Accuracy

Figure 5.3 and 5.4 show `diversity`, `coverage`, and accuracy of the SVM and FastText models as we vary the number of training examples for scenario-driven and paraphrase-based jobs, respectively. In this experiment, we use a combination of both generic and specific scenarios and paraphrasing examples.

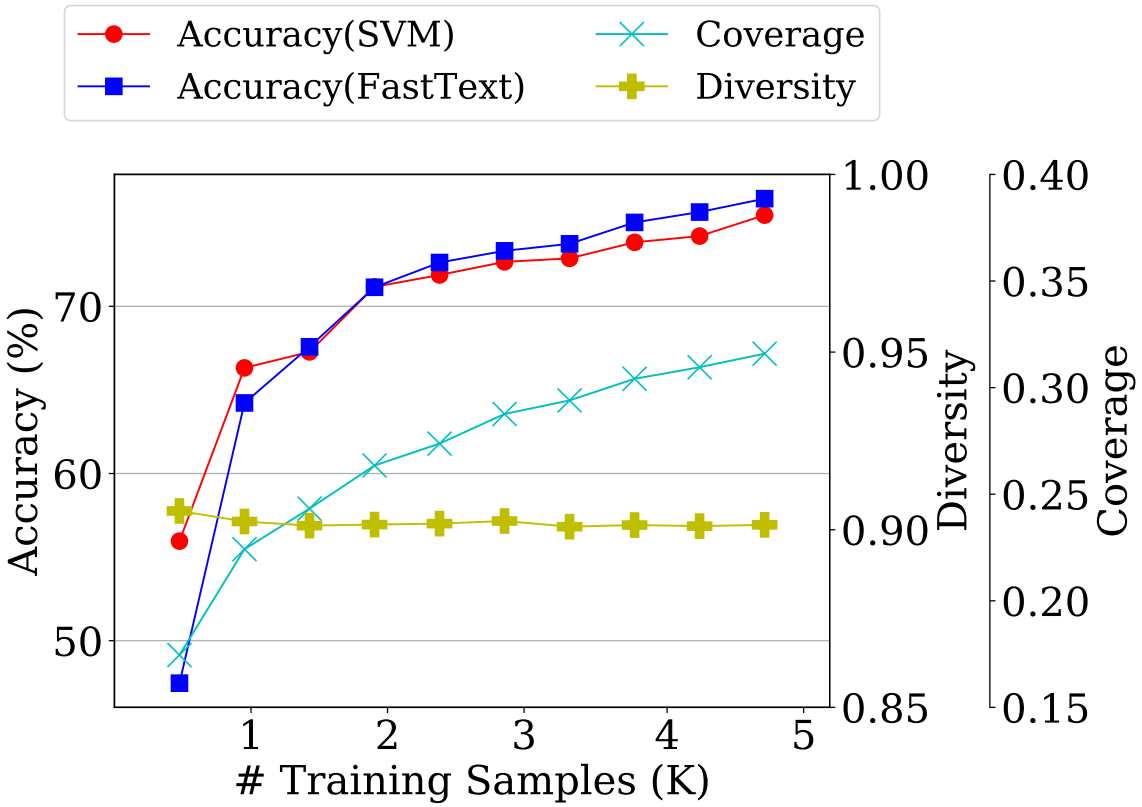



**Figure 5.4: Accuracy, `coverage` and `diversity` for paraphrasing jobs as the training data size increases. This data is collected using a combination of generic and specific paraphrase examples.**

We observe that for both scenario and paraphrase jobs, the `diversity` starts high ($> 0.90$) with a few hundred training samples and stay stable as training data size increases. This means that the new training examples generally have a low percentage of n-grams overlap and a long distance ($D(a, b)$) with the existing examples, therefore maintaining the overall high `diversity`. This indicates that the newly introduced examples are generally creative contributions from the crowd and not repeats or straightforward rephrase of the existing samples with the same words.

`coverage` starts low with a few hundred training examples and steadily increases as the training set grows. This indicates that the new training examples contain sentences that are semantically closer to the test set sentences than existing training examples, increasing the training set's scope to better cover the expression space

represented by the test set.

The accuracy of both SVM and FastText models follow a very similar trend to that of `coverage`, gradually increasing as more training samples are collected. The correlation between model accuracy and `coverage` shows that `coverage` is a more effective metric than `diversity` in evaluating the quality of a training set without training models.

We also observe diminishing returns in `coverage` as more data is collected. This trend roughly correlates with the diminishing return in accuracy of the SVM and FastText models. The trend in `coverage` provides insight into improvements in training data quality, which can inform the choice of when to stop collecting more data and start focusing on improving algorithms. This is further demonstrated by the way FastText consistently outperforms the SVM model when their accuracy and `coverage` of the training data saturate, indicating that the algorithm is the bottleneck for improving accuracy instead of the amount of training data.

**Key Insights** (1) `diversity` stays relatively constant with a high value as more training samples are collected, indicating that new distinct training examples are being introduced. (2) `coverage` continuously improves as data scales, showing that the training data is becoming more effective at covering the expression space defined by the test set. The trend of `coverage` closely correlates with the trend in model accuracy, indicating that `coverage` is an effective metric at evaluating training data quality without requiring model training.

### 5.4.2 Comparing Scenario and Paraphrase Based Collection and Their Variants

Table 5.2 summarizes the model accuracy, `coverage` and `diversity` of both scenario-driven and paraphrase-based jobs. We studied three variants for each job

| | | Accuracy | | | |
|---|---|---|---|---|---|
| Template | Type | SVM | FastText | CVG | DIV |
| Scenario | Generic | 68.49 | 69.70 | 0.30 | 0.90 |
| | Specific | 65.86 | 68.10 | 0.29 | 0.89 |
| | **Both** | **74.77** | **75.48** | **0.32** | **0.91** |
| Paraphrase | Generic | 68.60 | 70.50 | 0.30 | 0.88 |
| | Specific | 67.80 | 67.77 | 0.29 | 0.87 |
| | **Both** | **75.46** | **76.44** | **0.32** | **0.90** |

**Table 5.2: Accuracy, `coverage` and `diversity` for the six template + prompt conditions considered, all with ~4.7K training samples.**

type, where we use different mixtures of prompt type (generic prompts only, specific prompts only and combined prompts). All configurations are evaluated using training data of the same size (~4.7K) and on the same test set.

For both scenario and paraphrase jobs, using a mixture of both generic and specific prompts yields training data with higher `coverage` and models with higher accuracy than using only generic or specific prompts.

Table 5.2 compares scenario and paraphrasing jobs. As described in § 5.3.2, the paraphrasing examples were based on the scenario descriptions so we are only measuring the impact of different job types. The two approaches lead to similar results across all metrics. This shows that despite the instructions being distinctly different, scenario-driven and paraphrasing jobs generally yield training data of similar quality.

**Key Insights** (1) A mixture of generic and specific scenarios and paraphrasing examples yields the best training data given a fixed number of training examples, in terms of both `coverage` of the training set and the accuracy of the downstream models. (2) Scenario-driven and paraphrasing based crowdsourcing jobs yield similar quality training data despite having different job instructions and templates.

### 5.4.3 Sampling Prompts from the Test Set

We now investigate a different way to generate the prompts used for the crowdsourcing jobs. In the context of scenario-driven and paraphrasing jobs, prompts are

| | Accuracy | | | |
|---|---|---|---|---|
| | SVM | FastText | CVG | DIV |
| Manual generation | 75.46 | 76.44 | 0.32 | 0.90 |
| **Test set sampling** | **83.05** | **84.69** | **0.40** | **0.92** |

**Table 5.3: Comparison of manually generating prompts and sampling from test set, evaluated on half of the test data (kept blind in sampling).**

the scenario descriptions and the example sentences provided to workers to rephrase, respectively. In § 5.3.1 and 5.3.2, engineers manually generated the prompts based on the definition of each intent. While manual generation guarantees high quality prompts, it requires engineering effort and could potentially be biased by the engineer's perspective. One way to reduce such effort and bias is to automatically source prompts based on real user queries.

We divide the test set into two equal halves. For each intent, we randomly sample 5 utterances from the first half of the test set and use them as prompts to construct paraphrasing jobs. The second half of the test set is kept entirely blind and used for evaluation.

**Manual Generation vs. Test Set Sampling** Table 5.3 shows the accuracy, `coverage` and `diversity` of a training set collected with 4 manually generated paraphrasing examples vs. with 4 paraphrasing examples sampled from the first half of the test set. The accuracy for both methods is evaluated on the second half of the test set (kept blind from prompt sampling). The results show that sampling from the test set leads to a training set that has 8% higher `coverage`, 2% higher `diversity` and yields models with 8% higher accuracy, compared to manual generation.

**Varying the Number of Prompts** Table 5.4 shows the accuracy, `coverage` and `diversity` of training data collected using a varying number (1-5) of unique paraphrasing examples sampled from the test set. We observe that test set accuracy

| # of paraphrasing prompts | Accuracy SVM | FastText | CVG | DIV |
|---|---|---|---|---|
| 1 | 71.46 | 71.69 | 0.31 | 0.88 |
| 2 | 78.34 | 79.33 | 0.36 | 0.91 |
| 3 | 81.47 | 82.67 | 0.39 | 0.91 |
| 4 | 83.05 | 84.69 | 0.40 | 0.92 |
| **5** | **84.61** | **85.96** | **0.41** | **0.92** |

**Table 5.4: Accuracy, `coverage` and `diversity` of paraphrasing jobs using 1-5 prompts sampled from the test set, with constant training set size (~4.7K).**

improves as we use more unique prompts but eventually there are diminishing returns. Increasing the number of prompts from 1 to 2 increases the accuracy by 6.9% and 7.6% for SVM and FastText, respectively, while increasing the number of prompts from 4 to 5 improves their accuracy by only 1.6% and 1.3%.

## 5.5 Summary

Training data is the key to building a successful real-world intelligent dialogue system, and efficiently collecting large scale robust training data via crowdsourcing is particularly challenging. In this chapter, we introduce and characterize two training data quality evaluation metrics. We verify their effectiveness by training models of well-known algorithms and correlating the metrics with model accuracy. We show that an algorithm-independent `coverage` metric is effective at providing insights into the training data and can guide the data collection process. We also studied and compared a range of crowdsourcing approaches for collecting training data for a many-intent classification task for a dialogue system. Our observations provide several key insights that serve as recommendations for future dialogue system data collection efforts, specifically that using a mixture of generic and specific prompts and sampling prompts from the real user queries yields better quality training data.

# CHAPTER VI

# Conclusion

Intelligent applications are becoming increasingly more personal and knowledgeable and widely adopted as they are integrated by default on various mobile devices. However, there exists a series of challenges across the system stack in building an intelligent application system. This dissertation investigates the challenges in the compute and data components and propose system design and techniques to improve application and system performance.

To optimize computation performance, I first identify wireless communication as the bottleneck in the status-quo approach of cloud-only intelligent application processing in terms of end-to-end response latency and mobile energy consumption. I design a lightweight runtime scheduler that dynamically partitions neural network computation between the mobile device and the cloud to achieve low latency, low energy and high datacenter throughput, based on various factors including wireless network speed, neural network topology and datacenter load. Secondly, I characterize a suite of state-of-the-art deep learning based natural language processing applications and identify that they share a unique recurrent and dependent neural network computation pattern that render the existing GPU acceleration technique ineffective for this class of application. Leveraging this unique characteristic, I design and develop a novel software system to effectively accelerates this class of applications on the

GPU, where neural network inputs across queries are collected and being processed as a batch for high GPU throughput. For the data aspect of an intelligent application system, to address the challenge of effectively collecting large-scale high-quality training data for building high-accuracy machine learning models, I design and propose two novel metrics of evaluating training data quality. These metrics are designed to capture the semantic heterogeneity of the training data and how effective the training data is at representing the scope of the target task. Leveraging these metrics, I investigate multiple crowdsourcing methods and the quality of their corresponding training data and provide insights and recommendations on the best practices for crowdsourcing training data.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Apple moves to third-generation Siri back-end, built on open-source Mesos platform. http://9to5mac.com/2015/04/27/siri-backend-mesos/. Accessed: 2016-08.

[2] Apple's Massive New Data Center Set To Host Nuance Tech. http://techcrunch.com/2011/05/09/apple-nuance-data-center-deal/. Accessed: 2016-08.

[3] Baidu Supercomputer. https://gigaom.com/2015/01/14/baidu-has-built-a-supercomputer-for-deep-learning/. Accessed: 2016-08.

[4] Clipped summarizes anything into bullet points and infographics through the power of ai. http://clipped.me/. Accessed: 2016-11-18.

[5] Duc-2003. http://duc.nist.gov/data.html.

[6] Facebook github treadmill. https://github.com/facebook/treadmill. Accessed: 2016-10-11.

[7] Flipbord's approach to automatic summarization. http://engineering.flipboard.com/2014/10/summarization/. Accessed: 2016-11-18.

[8] Google Brain. https://backchannel.com/google-search-will-be-your-next-brain-5207c26e4523#.x9n2ajota. Accessed: 2017-01.

[9] Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html. Accessed: 2017-01.

[10] Microsoft Deep Learning Outperforms Humans in Image Recognition. http://www.forbes.com/sites/michaelthomsen/2015/02/19/microsofts-deep-learning-project-outperforms-humans-in-image-recognition/. Accessed: 2016-08.

[11] Movie review data. https://www.cs.cornell.edu/people/pabo/movie-review-data/.

[12] NVIDIA Jetson TK1 Development Kit: Bringing GPU-accelerated computing to Embedded Systems. Technical report. Accessed: 2017-01.

[13] Nvidia's Tegra K1 at the Heart of Google's Nexus 9. http://www.pcmag.com/article2/0,2817,2470740,00.asp. Accessed: 2016-08.

[14] TestMyNet: Internet Speed Test. http://testmy.net/. Accessed: 2015-02.

[15] The 'Google Brain' is a real thing but very few people have seen it. http://www.businessinsider.com/what-is-google-brain-2016-9. Accessed: 2017-01.

[16] Watts Up? Power Meter. https://www.wattsupmeters.com/. Accessed: 2015-05.

[17] Whitepaper: NVIDIA Tegra X1. Technical report. Accessed: 2017-01.

[18] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks. Fathom: reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016.

[19] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[20] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[21] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 247–257, New York, NY, USA, 2010. ACM.

[22] D. Chen and W. Dolan. Collecting highly parallel data for paraphrase evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 190–200, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[23] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[24] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014.

[25] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In

*Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[26] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014.

[27] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association.

[28] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[29] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and A. Ng. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[30] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[31] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 2011.

[32] A. Conneau, H. Schwenk, L. Barrault, and Y. LeCun. Very deep convolutional networks for natural language processing. *CoRR*, abs/1606.01781, 2016.

[33] M. G. Core and J. F. Allen. Coding dialogs with the damsl annotation scheme. In *Working Notes of the AAAI Fall Symposium on Communicative Action in Humans and Machines*, pages 28–35, Cambridge, MA, November 1997.

[34] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[35] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[36] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 92–104, New York, NY, USA, 2015. ACM.

[37] O. Feyisetan, E. Simperl, M. Luczak-Roesch, R. Tinati, and N. Shadbolt. An extended study of content and crowdsourcing-related performance factors in named entity annotation. *Semantic Web*.

[38] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently.

[39] C. Grady and M. Lease. Crowdsourcing document relevance assessment with mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pages 172–179, Los Angeles, June 2010. Association for Computational Linguistics.

[40] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 64–76. IEEE, 2016.

[41] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An execution framework for deep neural networks on resource-constrained devices. In *MobiSys*, 2016.

[42] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15, New York, NY, USA, 2015. ACM.

[43] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 27–40, New York, NY, USA, 2015. ACM.

[44] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[45] M. Iyyer, J. L. Boyd-Graber, L. M. B. Claudino, R. Socher, and H. Daumé III. A neural network for factoid question answering over paragraphs. In *EMNLP*, 2014.

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

[47] Y. Jiang, J. K. Kummerfeld, and W. S. Lasecki. Understanding task design trade-offs in crowdsourced paraphrase collection. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 103–109, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[48] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, Valencia, Spain, April 2017.

[49] G. Kazai, J. Kamps, and N. Milic-Frayling. An analysis of human factors and label accuracy in crowdsourcing relevance judgments. *Information Retrieval*, 16(2):138–178, 2013.

[50] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[52] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[53] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen. Accurate and Compact Large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, pages 662–665, 2013.

[54] X. Lei, A. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, pages 662–665, 2013.

[55] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, Dec. 2004.

[56] X. Liang, X. Shen, J. Feng, L. Lin, and S. Yan. Semantic object parsing with graph LSTM. *CoRR*, abs/1603.07063, 2016.

[57] B. Liu. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, 5(1):1–167, 2012.

[58] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM.

[59] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–381. ACM, 2015.

[60] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 301–312. IEEE Press, 2014.

[61] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[62] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. *ISPASS '12: International Symposium on Performance Analysis of Systems and Software*, April 2012.

[63] D. Meisner, J. Wu, and T. F. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 35–45. IEEE, 2012.

[64] A. Nikravesh, D. R. Choffnes, E. Katz-Bassett, Z. M. Mao, and M. Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *International Conference on Passive and Active Network Measurement*, pages 12–22. Springer, 2014.

[65] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[66] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11), 2015.

[67] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The kaldi speech recognition toolkit. In *Proc. ASRU*, 2011.

[68] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

[69] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium*

*on Computer Architecture*, ISCA '13, pages 24–35, New York, NY, USA, 2013. ACM.

[70] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[71] C. Rashtchian, P. Young, M. Hodosh, and J. Hockenmaier. Collecting image annotations using amazon's mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pages 139–147, Los Angeles, June 2010. Association for Computational Linguistics.

[72] J. Rogstadius, V. Kostakos, A. Kittur, B. Smus, J. Laredo, and M. Vukovic. An assessment of intrinsic and extrinsic motivation on task performance in crowdsourcing markets, 2011.

[73] A. M. Rush, S. Chopra, and J. Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[74] M. Sabou, K. Bontcheva, L. Derczynski, and A. Scharl. Corpus annotation through crowdsourcing: Towards best practice guidelines. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*. European Language Resources Association (ELRA), 2014.

[75] J. Schectman. Obama's Campaign Used Salesforce.com To Gauge Feelings of Core Voters. `http://blogs.wsj.com/cio/2012/12/07/obamas-campaign-used-salesforce-com-to-gauge-feelings-of-core-voters/`, 2012. [Online; accessed 2016-11-17].

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

[77] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[78] R. Snow, B. O'Connor, D. Jurafsky, and A. Ng. Cheap and fast – but is it good? evaluating non-expert annotations for natural language tasks. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 254–263, Honolulu, Hawaii, October 2008. Association for Computational Linguistics.

[79] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[80] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[81] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR)*, 2014.

[82] R. Vliegendhart, M. Larson, C. Kofler, C. Eickhoff, and J. Pouwelse. Investigating factors influencing crowdsourcing tasks with high imaginative load. In *Proceedings of the Workshop on Crowdsourcing for Search and Data Mining (CSDM) at the Fourth ACM International Conference on Web Search and Data Mining*, pages 27–30, 2011.

[83] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

[84] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[85] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 649–657, Cambridge, MA, USA, 2015. MIT Press.

[86] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference.