HARVARD UNIVERSITY

Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences
have examined a dissertation entitled:

"Computation-Cautious Machine Learning Systems"

presented by: Abdul Wasay

Signature _____
Typed name: Professor F. Doshi-Velez

Signature _____
Typed name: Professor D. Brooks

Signature _____
Typed name: Professor S. Idreos

Signature _____
Typed name: Professor A. Kumar

May 3, 2021

Computation-Cautious Machine Learning Systems

a dissertation presented
by
Abdul Wasay
to
The John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts
May 2021

Thesis advisor: Professor Stratos Idreos Abdul Wasay

Computation-Cautious Machine Learning Systems

Abstract

Deriving knowledge from data is central to how we live, learn, and decide: Machine learning and data science pipelines are extensively applied to extract knowledge from an ever-increasing amount of data across all fields including high-energy physics, astronomy, and genetics. These pipelines consist of multiple stages from data exploration to model design, training, and deployment. Different stages have their own set of algorithms and techniques, yet they share a common challenge – they involve repeated computation on huge data sets. This bottleneck slows down machine learning pipelines, which is problematic not only for latency-sensitive applications (such as self-driving cars and medical diagnosis), but as a result of this bottleneck, only a fraction of the generated data can be processed leading to lower quality models, fewer decisions per time unit, and overall, limited applicability of machine learning.

We introduce Computation-Cautious Machine Learning Systems – Data Canopy, Deep Collider, and MotherNets – that address the bottleneck of repeated computation and data movement across four critical stages of machine learning pipelines: (i) data exploration, (ii) model design, (iii) model training, and (iv) model deployment. During data exploration, *Data Canopy* enables reuse of computation and data movement across different statistical queries leading to several orders of magnitude (10× to 100×) improvement in the speed of data exploration and machine learning algorithms. *Deep Collider* reconsiders conventional model design wisdom and enables drastically better model design by balancing simultaneously accuracy, training time, deployment time, and memory resources. Finally, *MotherNets* enables fast and accurate training and deployment of ensembles of deep neural networks (2 to 3 percent reduced absolute test error rate and up to 35 percent faster training as compared to state-of-the-art approaches). MotherNets also establishes a new and navigable Pareto frontier for the accuracy-training cost tradeoff of deep neural network ensembles.

Contents

Acknowledgments

I want to begin by thanking my advisor, Stratos Idreos, first for believing in me and second for his gracious support and advice throughout my Ph.D. I grew tremendously as a writer, researcher, teacher, communicator, and person under his mentorship. I particularly enjoyed the many hours we spent learning about deep neural networks together or fine-tuning research manuscripts and conference presentations.

I am indebted to so many teachers and mentors who, over the years, expanded the way I look at the world. First and foremost, I would like to give a special thanks to Ihsan A. Qazi for his excellent class on data structures at LUMS, where this journey truly began, and for the countless ways in which he has been part of it since. I am incredibly grateful to: Nabiha Meher Shaikh, whose course at LUMS made me fall in love with the art of writing; Finale Doshi-Velez, whose machine learning course at Harvard serves both as an inspiration and background for this work; and teachers from various middle and high schools I attended in Porto Novo, Mbale, Lahore, and Rabwah who got me ready, bit-by-bit, for this journey.

None of this will be possible without the support of the excellent, funny, and eccentric people I spent my days at Harvard DASLab with: Mike Kester, Manos Athanasoulis, Lukas Maas, Brian Hentschel, Kostas Zoumpatianos, Niv Dayan, Wilson Qin, Kenneth Bøgh, Subarna Chatterjee, and Sanket Purandare. Thanks for countless brainstorming sessions, for being there through many paper rejections, and for, you know, hanging out. More broadly, I am thankful to many collaborators with whom I had the chance to work with and learn from: Alkis Smitsis, Xinding Wei, Zichen Zhu, Pablo Ruiz Ruiz, Yuze Liao, Sanyuan Chen, Neil Band, Chang Xu, Haochen Yang, Ziyi Guo, Longshen Ou, and Dhruv Gupta. Also, a very special thanks to Susan Welby for providing administrative support during my Ph.D.

Finally, I am thankful to my family and friends from all over for their support throughout this journey. Shukriya.

1

Introduction

1.1 The Machine Learning Era

Machine learning enables computers to perform tasks by learning from experience – i.e., data – instead of executing explicit programs. Over the past two decades, the ease of collecting and storing data coupled with the introduction of highly-specialized machine learning algorithms has enabled computers to efficiently and accurately perform many complicated tasks. Computers can now classify images at human accuracy, infer meaning from natural language, and help drive cars (Bengio et al. 2015). The future holds even more significant promise for machine learning with the increasing adoption of various machine learning methods in medicine, science, and society.

1.2 Machine Learning Happens in Pipelines

To perform any machine learning task, no matter how simple or complex, machine learning practitioners and data scientists assemble pipelines, which are made up of multiple stages (Sparks et al. 2017; Boehm et al. 2019): (i) Data exploration, (ii) Model

design, (iii) Model training, and (iv) Model deployment. These stages progressively convert raw data sets into machine learning models that are deployed to automate various decision-making processes.

We look at machine learning pipelines across three important scientific domains identifying characteristics of all four stages.

Example 1: Detecting Rare Events in High-Energy Physics Data. Machine learning pipelines are prevalent in High-Energy Physics, a data-intensive domain where particle accelerators (such as the Large Hadron Collider) generate petabytes of data every day. One such pipeline is deployed at the European Organization for Nuclear Research (CERN). This pipeline uses deep learning models to detect rare events in data generated by particle accelerators. Rare events have specific statistical signature and are of interest to particle physicists (Nguyen et al. 2019).

We find all four stages of machine learning pipelines here. This pipeline begins at the data exploration phase: Data scientists use filters and statistical properties to narrow down a training set that contains sufficient examples of rare and interesting events. The next step is to design a deep learning model that can learn to detect rare events from the training data set with high accuracy. To do this, data scientists use a sample of the training data set to choose between hundreds of deep learning models that range from fully-connected neural networks to convolutional and recurrent neural networks. The selected model is then trained on the entire training data set. The trained model is then deployed and it is periodically retrained to account for new data.

Example 2: Linking Cell Expression to Genetic Data. Genomics is another domain that employs machine learning pipelines. An important area of inquiry in genomics is to understand the relationship between genotype (genetic information stored in the DNA) and phenotype (the cells that make up an organism) (Moen et al. 2019). A machine learning pipeline is used to automatically learn this relationship from genetic data originating from various sensors (Culley et al. 2020).

In this machine learning pipeline, the first step (which falls under data exploration) involves correlation analyses between different streams of sensor data. The aim here is to discover which subsets of data provide the most information for model design and

training. In the next stage, i.e., model design, genomic researchers come up with custom neural network models called Multi-modal Artificial Neural Networks that can integrate information from various genetic data sources. This design process requires an extensive set of experiments on dozens of existing models. The finalized model is then trained and evaluated on various combinations of data sources and made available to the genomics research community.

Example 3: Discovering New Planets from Telescope Data. Finally, we look at a machine learning pipeline in Astronomy. In this field, a major new source of data is NASA's Kepler Space Telescope program. This program captures signatures of various planets in the solar system that have similar properties to planet Earth. The data it captures, however, is extremely large and noisy. By using deep learning pipelines identical to those used in image classification, astronomers recently discovered two new planets (Shallue and Vanderburg 2018).

We see all four stages at work in this machine learning pipeline: In the first step, through a combination of filtering and exploratory statistical analysis, data scientists synthesize the training data from three different data sources. During the design stage, data scientists search over the design space of various types of convolutional neural network models. During the training phase, they train multiple copies of the same selected model with different initializations to create an ensemble having various diversely-trained networks. Finally, during the deployment phase, outputs from every network of the ensemble are averaged together to produce a final set of predictions for candidate planets. These set of predictions are then verified by astronomers.

Machine Learning Pipelines are Everywhere. These pipelines are not exclusive to these three domains but we can find instances of these pipelines across all areas ranging from education to agriculture to finance (Boehm et al. 2019). Overall, these pipelines allow data scientists and experts from various domains to convert raw data into deployable models that can enable multiple forms of decision-making.

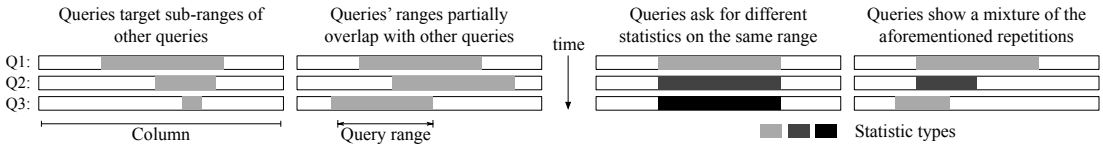


Figure 1.1: In exploratory statistical analysis, queries request for a given statistic on a given data range and show various forms of repetition.

1.3 Bottleneck: Repeated Computation and Data Movement

All stages of machine learning pipelines have their own set of algorithms and techniques. However, they suffer from a shared challenge: They involve repeated computation on huge data sets. This bottleneck slows down all stages of machine learning pipelines. Slow machine learning pipelines are problematic as many applications, such as medical diagnosis and self-driving cars, require quickly processing new data and incorporating it into deployed models. For instance, in one use case, where deep learning models are applied to detect Diabetic Retinopathy (a leading cause of blindness), newly-labeled images become available every hour. Thus, incorporating new data in the neural network models as quickly as possible is crucial to enable a more accurate diagnosis for the immediately next patient (Gulshan et al. 2016). Additionally, slow machine learning pipelines limit machine learning's reach as only application scenarios with a significant amount of compute and memory resources can feasibly use these pipelines.

We characterize the sources of this bottleneck, i.e., repeated computation and data movement, across the different stages of machine learning pipelines:

1.3.1 Data Exploration

Data Exploration and Statistics. Machine learning pipelines begin with a data exploration phase where statistics play an essential role (Surajit 2016). During this phase, data scientists develop an initial understanding of the data by using statistics to summarize variables within the data set, understand trends in variables, and correlate these trends with those of other variables (Guo 2012; Madigan and Wasserstein 2013). For instance, variance in seismic activity of an area represents how prone it is

to earthquakes, and correlations between seismic measurements across various sensors help to predict future patterns of seismic activity (Williams et al. 2010). Moreover, statistics – such as mean, variance, and correlations – serve as building blocks of core machine learning classification and filtering algorithms such as simple linear regression, Bayesian classification, and collaborative filtering (Bishop 2006). Overall, statistical analysis forms the staple of data exploration across all fields.

Bottleneck: Repetitive Calculation of Statistics. Exploratory statistical analysis, a typically unstructured procedure, results in the repetitive calculation of statistics. Every result provides data scientists with knowledge and cues for what to ask next or which model to try out. Different statistics are successively calculated on the same part of the data. Even the exact statistics are recomputed with varying resolution and on data ranges (data portions) that overlap with previously accessed data ranges. Effectively an exploration session consists of numerous such repeated queries until a pattern is found (Idreos et al. 2015). Figure 1.2 summarizes different forms of such repetitive access patterns. Overall, these repetitive access patterns result in repeated computation and data movement, slowing down the process of exploratory analysis.

1.3.2 Model Design

Designing Deep Learning Models under a Resource Budget. The next step in the machine learning pipeline is model design, i.e., deciding on what model to use for a specific application scenario. Modern machine learning pipelines increasingly use deep learning models to capture complex patterns in data sets. Designers of these deep learning models navigate a complex design landscape: To start off, they decide on a network architecture to use in their model. Then, they have to consider whether to use a single network or build an ensemble model with multiple networks. Additionally, they have to decide how many neural networks to use as well as their individual designs i.e., to find a desirable configuration of depth, width, and number of networks in their model. Modern applications with diverse requirements further complicate these decisions as what is desirable varies. Facebook, for instance, requires convolutional neural network models that strike specific tradeoffs between accuracy and inference time across 250

different types of smartphones (Wu et al. 2019). As a result, not just accuracy but a diversity of metrics – such as inference time and memory usage – inform whether a model gets used (Sze et al. 2017b).

Bottleneck: Lack of a Robust and Holistic Assessment. There is no holistic empirical or theoretical framework to consistently analyze the relationship between a neural network design (with a given configuration of depth, width, and number of networks) and various metrics of interest such as accuracy, training time, and memory usage. As a result model designer rely on incomplete conventional wisdoms that result in either a series of sub-optimal designs or expensive hit-and-trial exploration of the design space. Both of these outcomes ultimately result in repeated computation and data movement as multiple models have to be trained and analyzed every single time.

1.3.3 Training and Deployment

Training and Deploying Neural Network Ensembles. The next step in machine learning pipelines is to train and deploy the machine learning model. Here, various applications increasingly train and deploy ensembles of multiple neural networks. This is because ensembles function as collections of experts and have been shown, both theoretically and empirically, to improve generalization accuracy (Russakovsky et al. 2015). For instance, deep neural network ensembles predict relationships between chemical structure and reactivity (Agrafiotis et al. 2002), segment complex images with multiple objects (Ju et al. 2017), and are used in zero-shot as well as multiple-choice learning (Guzman-Rivera et al. 2014; Ye and Guo 2017). Further, several winners and top performers on the ImageNet challenge are ensembles of neural networks (Lee et al. 2015a; Russakovsky et al. 2015).

Bottleneck: The Growing Training and Deployment Cost. Using ensembles of multiple deep neural networks takes a prohibitively large amount of time and computational resources. Even on high-performance hardware, a single deep neural network may take several days to train. This training cost grows linearly with the ensemble's size as every neural network in the ensemble needs to be trained (Szegedy et al. 2015; He

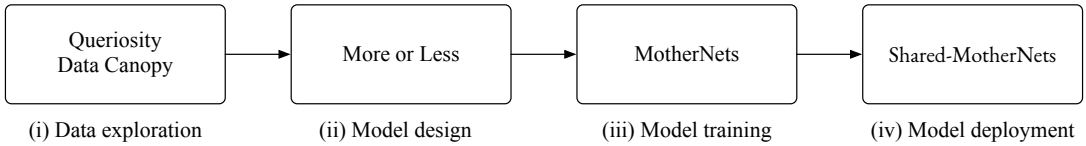


Figure 1.2: We design computation-cautious machine learning systems that address the bottleneck of repeated computation and data movement across all stages of machine learning pipelines.

et al. 2016; Huang et al. 2017b,a). Similarly, during deployment, we need to infer from every member of the neural network ensemble. The rising cost boils down to repeated computation and data movement: During training, multiple different neural networks need to be trained. During deployment, every data item needs to be passed through all of the neural networks in the ensemble.

1.4 Computation-Cautious Machine Learning Systems

We design Computation-Cautious Machine Learning Systems that address the growing bottleneck of repeated computation and data movement across all stages of machine learning pipelines: Data Canopy accelerates the process of exploratory statistical analysis by reusing computation and data movement between different statistical queries. Deep Collider demystifies the process of model design through an empirical framework to holistically analyze the design space of neural network models. Finally, MotherNets enable rapid training and deployment of neural network ensembles by sharing computation and data movement between different neural networks in an ensemble.

1.4.1 Data Canopy: Reuse During Data Exploration

We take a step to address the bottleneck of repeated computation and data movement during data exploration through Data Canopy, where statistics are synthesized from a library of basic aggregates computed and managed over sub-ranges of the data set. Data Canopy enables the reuse of these basic aggregates across overlapping parts of the data set and between different types of statistics. Data Canopy can be populated ahead

of time (with a single pass over the data set), during the analysis phase itself, or in an opportunistic manner. Future queries can avoid having to repeatedly go back to the base data but instead can synthesize statistics from Data Canopy. Compared to state-of-the-art tools (such as NumPy and Modeltools) that provide static and slow performance, the performance of Data Canopy keeps on improving as future queries can use past computation and data access leading to *multiple orders of magnitude improvement in the speed of exploratory workloads and statistics-based machine learning algorithms.*

1.4.2 Deep Collider: Demystifying Machine Learning Model Design Under a Parameter Budget

We develop a detailed and extensive experimental framework to isolate the impact of the critical design knobs: (i) depth, (ii) width, and (iii) number of networks, on all relevant metrics: (i) accuracy, (ii) training time, (iii) inference time, and (iv) memory usage. Crucially the number of parameters is a control knob in this framework and remains fixed across results that can be studied together. We apply this framework to a critical part of the design space that is not well-understood: That is how to decide between the alternatives of expanding a single network model or increasing the number of networks and using them together in an ensemble. This framework *questions and expands the conventional wisdom* in deep learning model design: We show that under a parameter budget, ensembles can provide not only better accuracy than a single model but can also train and deploy faster while using far less memory

1.4.3 MotherNets: Sharing Computation During Model Training and Deployment

During the training and deploying of deep neural network ensembles, we address this bottleneck of repeated computation and data movement through MotherNets. The high-level idea is to first capture the structural similarity present in a given ensemble in the form of one or more MotherNets and then to train or infer from these MotherNets only once. During training, instead of every network, only the MotherNets are trained from scratch. What is learned by the trained MotherNets is then transferred to every member

of the ensemble; Every network only requires a minor amount of incremental training. MotherNets, in a similar fashion, improves deployment, where parameters originating from the MotherNets are tied together during training. This yields an ensemble with a single copy of MotherNets parameters reducing the inference time and memory requirement significantly. MotherNets is the first general-purpose fast ensemble training and deployment technique that extends to ensembles of diverse architectures as opposed to state-of-the-art approaches that generate ensembles from a monolithic architecture. MotherNets also establishes a new Pareto frontier for the accuracy-training cost tradeoff of deep neural network ensembles and provides *the most accuracy for the least amount of training time* when compared to all other ensemble training techniques.

1.4.4 Published Papers

The material in this thesis has been the basis for a number of publications in major refereed data systems and machine learning venues.

1. Abdul Wasay, Manos Athanassoulis, and Stratos Idreos, *Queriosity: Automated Data Exploration*, in Proceedings of the IEEE International Congress on **Big Data**, 2015 (Wasay et al. 2015).

2. Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos, *Data Canopy: Accelerating Exploratory Statistical Analysis*, in Proceedings of the ACM **SIGMOD** International Conference on Management of Data, 2017 (Wasay et al. 2017).

3. Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos, *MotherNets: Rapid Deep Ensemble Learning*, in Proceedings of the Conference on Machine Learning and Systems **MLSys**, 2020 (Wasay et al. 2020).

4. Abdul Wasay and Stratos Idreos, *More or Less: When and How to Build Convolutional Neural Network Ensembles*, in Proceedings of the International Conference on Learning Representation **ICLR**, 2021 (Wasay and Idreos 2021).

5. Abdul Wasay, Subarna Chatterjee, and Stratos Idreos, *Deep Learning: Systems and Responsibility*, in Proceedings of the ACM **SIGMOD** International Conference on Management of Data, 2021 (Wasay et al. 2021).

1.5 Thesis Outline (How to Read)

We organize the rest of the thesis as follows. First, Chapter 2 provides background on various stages of machine learning pipelines. In specific, we provide an overview of (i) data exploration and the role played by statistics in this process, (ii) fundamentals of deep neural networks and how to train and deploy them, (iii) basics of neural networks ensembles, and (iv) how we can transfer what is learned by one model to another. Chapter 3 positions Computation-Cautious Machine Learning Systems with respect to related work from data systems and machine learning communities. We outline the novelty this thesis brings on top of existing systems and frameworks. After reading Chapter 3, the rest of the chapters can be read independently. Chapter 4 through Chapter 6 describe the three Computation-Cautious Machine Learning Systems: Data Canopy, Deep Collider, and MotherNets. We provide detailed system design and algorithms. We also extensively evaluate these systems and show how they advance state-of-the-art techniques in their respective stages of machine learning pipelines. Finally, in Chapter 7, we conclude and discuss primary future research goals for Computation-Cautious Machine Learning Systems.

2

Background

2.1 Data Exploration and Statistics

Many machine learning pipelines across different fields begin with a data exploration phase (Surajit 2016). During this phase, data scientists develop an initial understanding of the data by using statistics to summarize variables within the data set, understand trends in variables, and correlate these trends with those of other variables (Guo 2012; Madigan and Wasserstein 2013). For instance, variance in seismic activity of an area represents how prone it is to earthquakes, and correlations between seismic measurements across various sensors help to predict future patterns of seismic activity (Williams et al. 2010). Moreover, statistics – such as mean, variance, and correlations – serve as building blocks of core machine learning classification and filtering algorithms such as simple linear regression, Bayesian classification, and collaborative filtering (Bishop 2006). Overall, statistical analysis forms the staple of data exploration across all fields.

	Exact repetition	Template repetition	Column set repetition
SQLShare	54.65%	36.93%	4%
SDSS	99.8%	99.7%	97%

Table 2.1: Exploratory workloads in the sciences exhibit high repetition in queries.

2.1.1 Characterizing Exploratory Workloads

Exploratory statistical analysis, a typically unstructured procedure, results in the repetitive calculation of statistics. Every result provides data scientists with knowledge and cues on what to ask next or which model to try out. By query here, we mean a request to compute a given statistic over a given data part. Different statistics are successively computed on the same part of the data. Even the exact statistics are recomputed with varying resolution and on data ranges (data portions) that overlap with previously accessed data ranges. Effectively an exploration session consists of numerous such repeated queries until a pattern is found (Idreos et al. 2015).

Repetition appears in various forms in various workloads. Table 2.1 shows the repetition in two publicly available workloads: SDSS SkyServer (Foundation) and SQLShare (Howe et al.). These workloads are composed of both handwritten and computer-generated SQL queries. Up to 97 percent of the queries repeat at least once in SDSS. Queries repeat less frequently in the SQLShare workload. However, up to 55 percent of queries still target a non-distinct set of columns. Furthermore, studies show that repetition is higher in interactive exploratory analysis (Jain et al. 2016).

2.1.2 Data Science Tools and Statistics

Data scientists have a spectrum of tools available to them for exploratory statistical analysis. At one end, this spectrum includes software libraries, such as NumPy (num 2013) and Modeltools (Foundation 2016), with flexible functionality but no in-built data management. On the other end of this spectrum are highly-optimized relational database systems but with limited statistical functionality. Database connectors like SciDB-Py (SciDB 2016), MonetDB.R (Mühleisen and Lumley 2013), and Psycopg (psy

2014) connect a database backend with a flexible language, thereby providing a good compromise between flexibility and data management. Such connectors offer the primary benefit of computing statistics inside the database system without moving data.

2.2 Deep Learning

Deep learning has seen an increasing amount of success thanks to deep neural networks, a set of powerful computational models that can learn intricate and complex patterns directly from data (LeCun et al. 2015). Researchers across several research communities can now solve problems that had evaded them for decades by applying deep neural networks. For instance, by applying deep learning models, researchers can localize and label objects within an image 3× more accurately than state-of-the-art classical methods (Liu et al. 2020). These intricate computational models have come to power countless aspects of our society today: Deep neural networks can translate languages, help drive cars, and diagnose various diseases (Kim 2014; Dong et al. 2015; Niepert et al. 2016; Shen et al. 2017).

The concept of deep learning has been around for over half a century. Researchers introduced neural networks in the forties and developed algorithms to train them in the sixties (McCulloch and Pitts 1943; Rosenblatt 1958; Widrow et al. 1960). In the past decade, however, three complimentary trends have catapulted deep learning into the academic and industrial limelight: First, hardware capabilities have improved remarkably (Sze et al. 2017a). With massive data parallelism, GPUs and TPUs can now train and deploy deep neural networks faster than ever before. Second, we can collect, store, and manage data at an unprecedented scale (Roh et al. 2019). Innovation in sensor technology, storage media, and database systems has significantly reduced the cost of obtaining data to feed deep neural networks. Finally, all key players in the computer science industry developed and open-sourced software libraries to design, train, and deploy deep neural networks (Nguyen et al. 2019). As such, researchers and practitioners in any field of human knowledge can quickly prototype and use deep learning pipelines. The future holds even more promise as researchers and practitioners develop bespoke neural network architectures and cultivate an ecosystem of hardware and software to better support existing and emerging deep learning architectures.

2.2.1 Deep Neural Networks

Deep neural networks are the workhorse of deep learning. They are capable of learning arbitrary functions from massive data sets. For instance, in classification tasks, deep neural networks can learn the mapping between data items and their labels, whereas, in sequence prediction tasks, they can learn how a given sequence relates to the sequence that follows it. What makes deep neural networks extremely powerful is that they automatically discover an appropriate representation from a large amount of training data for a given task. Data scientists can apply deep neural networks to new and arbitrary tasks, for which expert knowledge is either lacking or unscalable.

We can see an exciting example of deep neural network's versatility in how they are applied to diagnose Diabetic Retinopathy (a leading cause of blindness) (Kanungo et al. 2017). In this domain, experts are in short supply and are concentrated in a few geographical areas. However, they can conveniently label images gathered from various patients across the world. These labeled images are then used to train convolutional neural networks that are then deployed in mobile phones. Now, anyone with a mobile device can take a picture of a patient's eye and screen for Diabetic Retinopathy.

Deep neural networks today have dozens of layers consisting of non-linear modules. Every layer transforms its input from one level of representation to a more abstract level, which better captures aspects of the data set that are meaningful for a classification or detection task. For instance, when classifying images, the first layer may capture an image's low-level properties, such as the presence or absence of edges or dots. The second layer combines these properties to detect motifs such as a collection of edges. The third layer can assemble these motifs further, and the following layers can detect even more complex combinations of motifs (Zeiler and Fergus 2014; LeCun et al. 2015). Crucially, deep neural networks can learn these representations using a general-purpose training procedure.

We introduce various concepts related to the fundamental structure, training, and deployment of neural networks. We align our notation with that of Goodfellow et al. (2016). We summarize this notation in Table 2.2.

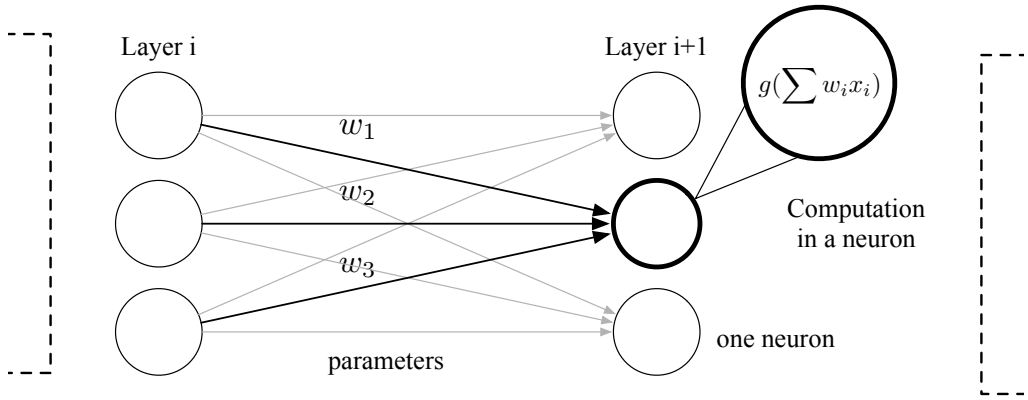


Figure 2.1: Neural networks are made up of layers of neurons, each neuron takes as input a subset of neurons from the previous layer and applies a set of weights to them.

2.2.2 Fundamentals of Deep Neural Networks

Neuron. A neuron is the fundamental building block of a neural network. Every neuron has an associated set of parameters or weights and, optionally, a non-linearity or activation function. Figure 2.1 depicts two layers of a neural network, where one of the neurons is highlighted in bold. The set of weights and the activation function of the highlighted neuron are $\{w_1, w_2, w_3\}$ and $g(\cdot)$ respectively. A neuron takes as input one or more neurons' output (or if the neuron is from the first hidden layer, it takes as input some region from within the input data item such as one or more pixels from an image) and computes an output. The output of a neuron is a function of its weights and the inputs it receives.

Layer. A layer in a neural network is composed of a set of neurons. The neurons in a certain layer interact with output from the previous layer. The output of this layer feeds into the next layer. In general, a neural network consists of an input layer, an output layer, and one or more hidden layers. Figure 2.1 depicts two layers (labelled layer i and layer $i+1$) with three neurons each.

Receptive Field. Every neuron in a layer has an associated receptive field, which is the subset of neurons from the previous layer fed as input to that neuron. In Figure 2.1, the receptive field of the highlighted neuron consists of every neuron from the previous

Description	Notation
Training data set	$\boldsymbol{X}^{(train)} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$; $\boldsymbol{Y}^{(train)} = \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m)}\}$
NN parameters	$\boldsymbol{\theta}$
Function of a NN	$f(\cdot, \boldsymbol{\theta})$
Training predictions	$\hat{\boldsymbol{Y}}^{(train)} = \{\hat{\boldsymbol{y}}^{(1)}, ..., \hat{\boldsymbol{y}}^{(m)}\}$
Empirical loss	$J(\boldsymbol{\theta})$
Test data set	$\boldsymbol{X}^{(test)} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$; $\boldsymbol{Y}^{(test)} = \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m')}\}$
Trasnfer data set	$\boldsymbol{X}^{(tran)} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m^*)}\}$; $\boldsymbol{Y}^{(tran)} = \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m^*)}\}$

Table 2.2: List of notations used throughout this chapter.

layer. This neuron is an example of a fully-connected neuron. In some cases, this connectivity can be sparser, as is the case with convolutional neurons discussed later on.

Neural Network Parameters. For a given neural network, the superset of all weights associated with all of its neurons are called its parameters.

Neural Network Architecture. We refer to the overall setting of a neural network as its architecture. A neural network architecture consists of various factors, including the number of layers, number of neurons in every layer, types of neurons, activation functions, and connectivity patterns between layers. Researchers and deep learning practitioners design dozens of new architectures every year to suit specific application needs and improve upon existing architectures' efficiency and quality. For instance, VGGNets, ResNets, Wide ResNets, and DenseNets are examples of different neural network architectures designed for image classification in the last decade.

2.3 Deep Learning and Classification

Classification is a fundamental problem in machine learning that has captured the attention of multiple generations of researchers across multiple fields. Given a data item, classification is concerned with determining which of a predefined set of classes the data item belongs to. Spam detection is a canonical classification problem, where given a bunch of emails, we are interested in putting them in one of two classes – spam or not

spam. Similarly, another example is from computer vision, where we are interested in determining the class to which an image belongs.

Generally speaking, machine learning classification is concerned with assigning labels to data belonging to a sample space. Formally, the sample space takes the form $X \times Y$, where X is the space of data and Y is the space of labels. $\boldsymbol{x} \in X$ is a data item of an arbitrary form, whereas $y \in Y$ represents one of k class labels $\{1, 2 \ldots k\}$. We are interested in using a machine learning model to learn the mapping between X and Y from a training data set. For instance, in CIFAR-10, a popular benchmark data set for image classification, $\boldsymbol{x}$ is an image, and $\boldsymbol{y}$ is the set of corresponding labels, i.e., one of the ten categories of everyday objects (cars, dogs, cats, etc.) (Krizhevsky 2009). Figure 2.2 shows sample images belonging to each of the ten classes in the CIFAR-10 dataset.

Recently, deep neural networks have made great strides in providing close-to-human performance on various classification tasks ranging from sentiment analysis to computer vision. The advantage that deep neural network models bring to classification is that they automatically infer features from labeled data during training. Consider the case of image classification. Before deep learning, computer vision researchers tried all sorts of things ranging from geometric methods that automatically detect edges and shapes to hand-engineered features (where an expert specifies aspects of images that they think are consequential to the classification task) (Huang 1996). Deep learning revolutionized image classification. Convolutional neural networks operating on huge data sets can automatically discover aspects or features of an image – such as dots, edges, and shapes – that are important for a given classification task.

2.3.1 Training Deep Neural Networks for Classification Tasks

Once a deep neural network architecture is specified, the training process is composed of alternating forward and backward passes until a specified metric (usually training accuracy) converges. Algorithm 2.1 provides an overview of this training process. Overall, during the training process, a set of labeled inputs is provided to the neural network. Based on this labeled data set, the neural network iteratively adjust its weights to learn a mapping function between the data and its labels.

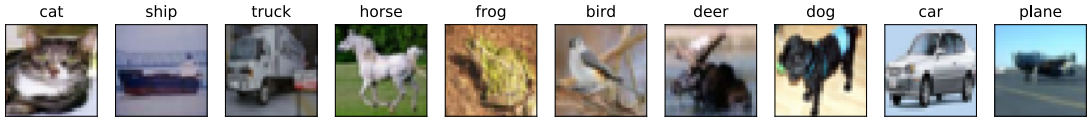


Figure 2.2: The CIFAR-10 dataset consists of images of everyday object. Each image is of size 32 × 32 pixels and has one of ten labels attached to it.

Training Data Set. The training data set, also referred to as the labelled data set, consists of m ordered data items $\boldsymbol{X}^{(train)} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ and corresponding labels $\boldsymbol{Y}^{(train)} = \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m)}\}$. Here, $\boldsymbol{x}^{(i)}$ is a vector of arbitrary length and $\boldsymbol{y}^{(i)}$ is a vector having as many components as labels in the data set and the component corresponding to the label of $\boldsymbol{x}^{(i)}$ is set to 1 and the rest of the components are set to 0, i.e., the labels are encoded using their one-hot representation. The CIFAR-10 dataset, for instance, consists of $50K$ training images. An image belongs to one of ten classes.

Forward Pass. Given a neural network architecture with parameters $\boldsymbol{\theta}$, the forward pass executes the neural network from the input layer to the output layer. Initially, the parameters $\boldsymbol{\theta}$ are initialized in a random manner. The forward pass proceeds by sequentially applying all parameters associated with every layer to the previous layer and feeding its output to the next layer. This results in a set of predictions $\hat{\boldsymbol{Y}}^{(train)} = \{\hat{\boldsymbol{y}}^{(1)}, ..., \hat{\boldsymbol{y}}^{(m)}\}$ corresponding to items in $\mathbf{X}^{(train)}$. Here, $\hat{\boldsymbol{y}}^{(i)} = f(\mathbf{x}^{(\mathbf{i})}, \boldsymbol{\theta})$, where $f(\cdot, \boldsymbol{\theta})$ is the functional mapping of the neural network.

Empirical or Training Loss. The empirical or the training loss $J(\hat{\boldsymbol{Y}}^{(train)}, \boldsymbol{Y}^{(train)})$ quantifies how different the predictions $\hat{\boldsymbol{Y}}^{(train)}$ are from the labels (or the ground truth) $\boldsymbol{Y}^{(train)}$. This loss can take various forms depending on the problem at hand and is usually differentiable with respect to the parameters $\boldsymbol{\theta}$.

Backward Pass. During the backward pass, based on the quality of the prediction, the weights associated with the neural network are adjusted. This step usually involves an optimization method such as gradient descent. In gradient descent, first, we compute the gradient of the training loss $J(\boldsymbol{\theta})$ with respect to the parameters. Then, every parameter in $\boldsymbol{\theta}$ is adjusted in the opposite direction of the gradient. The learning rate

Algorithm 2.1 Generalized deep neural network training procedure

Input: $\boldsymbol{X}^{(train)}, \boldsymbol{Y}^{(train)}, f(\cdot)$;
Initialize: $\boldsymbol{\theta}$;

while *network does not converge* **do**
```
// Execute forward pass
```
$\hat{\boldsymbol{Y}}^{(train)} \leftarrow f(\boldsymbol{X}^{(train)}, \boldsymbol{\theta})$
```
// Quantify the empirical loss
```
$J(\boldsymbol{\theta}) \leftarrow J(\hat{\boldsymbol{Y}}^{(train)}, \boldsymbol{Y}^{(train)})$
```
// Compute gradient
```
$G(\boldsymbol{\theta}) \leftarrow \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$
```
// Update parameters
```
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot G(\boldsymbol{\theta})$
end

return $f(\cdot, \boldsymbol{\theta})$

η quantifies how aggressively this adjustment takes place.

Convergence. A neural network is trained through a sequence of alternating forward and backward passes until a convergence criterion is met. There are various convergence criteria introduced in research. A simple criterion is to continue training until specified training rounds have been exceeded or a specific training loss has been achieved. Another criterion is to train until further training does not significantly alter the neural network parameters or improve its training loss.

2.3.2 Evaluation Metrics for Deep Neural Network Training

Metrics to evaluate deep neural networks fall under two categories: (i) Those related to quality, e.g., accuracy and robustness, and (ii) those related to the amount of resources required, e.g., training time, inference time, and memory usage. Modern deep learning pipelines are designed to optimize for one or a combination of these two sets of metrics. A primary goal is to achieve as high generalization accuracy as possible without incurring a significant increase in the training or inference cost.

Generalization Accuracy. When training a deep neural network, we would like to achieve as high as possible generalization accuracy, i.e., maximize its performance on a data set that the neural network has not seen before. This data set is still from the same sample space $X \times Y$ as the training data set. For instance, in our running example of the CIFAR-10 data set, we are interested in how well a neural network can classify images that are not present in – but are similar to those in the training data set. To quantify the generalization accuracy of a neural network, we use a test data set $\boldsymbol{X}^{(test)} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$ and corresponding labels $\boldsymbol{Y}^{(test)} = \{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m')}\}$. The generalization accuracy is defined as the proportion of the test data set that is correctly classified by the neural network:

$$Acc = \frac{\sum_{i=1}^{m'} \mathbb{1}(\arg\max(\boldsymbol{y}^{(i)}) = \arg\max(f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})))}{m'}$$

Here, $\mathbb{1}(.)$ is an indicator function that outputs 1 when its input is true and 0 otherwise. We use $\arg\max$ to extract the label from both the ground label vector $\boldsymbol{y}^{(i)}$ and the prediction vector $f(\boldsymbol{x}^{(i)}, \boldsymbol{\theta})$. We will also refer to the above metric as the test accuracy. The additive inverse of the generalization accuracy is called the generalization error (also referred to as the test error).

Training Time Per Epoch. Given hardware and data set, the training time per epoch is the amount of time it takes to complete one round of training of a neural network, i.e., one forward and one backward pass, on the entire training data set.

Total Training Time. The total training time is how long it takes for a neural network to converge, given the convergence criterion. We can express this time as the product of the number of epochs it takes to converge and the time taken per epoch.

Memory Usage. The memory a given neural network requires to train is another important metric. It is the total GPU and CPU memory needed to store the network parameters, training data set, and intermediate feature maps.

Inference Time. Once a trained neural network is deployed on particular hardware, the amount of time it takes to classify one data item is known as its inference time.

2.4 Convolutional Neural Networks

Next, we describe convolutional neural networks that are widely used in computer vision. Convolutional neural networks (ConvNets) are a particular class of deep neural networks that consider the spatial structure present in images and videos. By doing so, ConvNets are both more efficient and accurate at classifying images when compared with fully-connected neural networks. In particular, there are two significant ways in which ConvNets are tailored toward images and videos. First, layers in ConvNets have neurons arranged in three dimensions (width, height, and depth) to mimic the three-dimensional nature of image data sets. Second, ConvNets have two characteristic layers known as convolutional layers and pooling layers that take advantage of the fact that images have spatially-localized structure, i.e., images are not bags of pixels but rather that the position of pixels is important.

2.4.1 Three-Dimensional Layers

Images are three-dimensional grids with spatial dimensions (width and height) and depth (i.e., RGB values). In CIFAR-10, for example, every image has spatial dimensions 32×32 and a depth of 3 corresponding to the three RGB channels. ConvNets mimic this image structure and have three-dimensional layers having width, depth, and height. Effectively, each layer is a volume of neurons that operates on a volume of inputs and produces a volume of outputs. The intermediate outputs in a ConvNet are also referred to as feature maps as they capture features from images that are consequential to the computer vision task, i.e., image classification or object detection.

2.4.2 Types of Layers in ConvNets

ConvNets are composed of four different types of layers: convolutional, RELU, pooling, and fully-connected. Figure 2.3 shows a ConvNet with one example of each of these layer types. Figure 2.4 depicts the VGGNet architecture (a seminal convolutional neural network) composed of several sequences of all four layers. We describe every layer type.

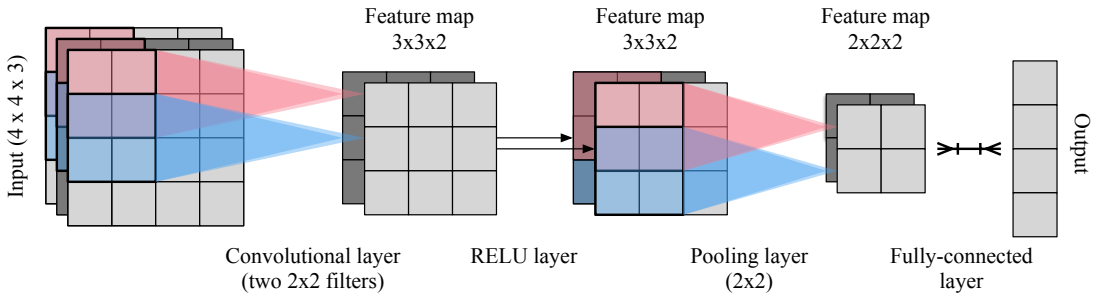


Figure 2.3: Convolutional neural networks are made up of various types of layers, each designed to serve a specific function.

Convolutional Layer. Convolutional layers have spatially connected neurons, i.e., neurons in a convolutional layer are only connected to a subset of neurons from the previous layer. Every neuron in this layer has a receptive field in the form of a cuboid, i.e., it operates on a small volume that extends through the full depth of the previous layer. The receptive fields of different neurons can have different degrees of overlaps between them controlled by the stride length. Every neuron computes the dot product between its receptive field and weights, producing a single numerical value as is the case for neurons in a fully-connected network. Typically, a convolutional layer has multiple stacks of neurons. Every stack of neurons produces a stack of outputs. The depth of the outputs produced is equal to the number of stacks. In Figure 2.3, the convolutional layer has neurons with a square receptive field of size 2×2, strided with a stride length of one, and there are two stacks of such neurons. Overall, this convolutional layer produces an output layer with depth two.

RELU Layer. RELU (Rectified Linear Unit) is one of the most popular activation function for neural networks. ReLU outputs the input directly if it is positive; otherwise, it provides zero as the output:

$$ReLU(x) = max(0, x)$$

The RELU layer applies this function to every element in its input layer, keeping both the spatial dimension and the depth intact.

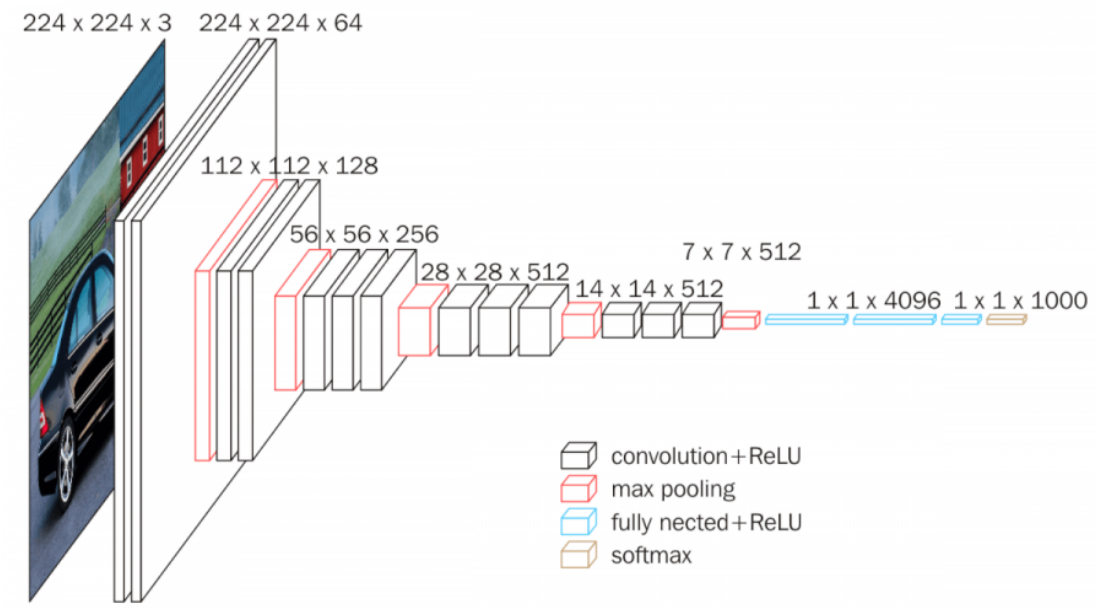


Figure 2.4: The VGGNet architecture is composed of a sequence of convolutional, ReLU, and pooling layers. The input image is successively transformed into a smaller more semantically meaningful form.

Pooling Layer. This layer reduces the spatial dimension (width and depth) of its input layer while preserving the depth. Sometimes, we also refer to it as the downsampling layer. Given a pooling function, a stride length, and a spatial receptive field, the pooling layer operates independently on every input layer's depth. It applies the pooling function to every box of the given receptive field strided with the given stride length. The pooling function is a non-linear downsampling function such as average or max. Figure 2.3 shows a pooling layer having neurons with a receptive field of size 2×2 and unit stride length.

Fully-Connected Layer. One or more fully-connected layers appear toward the end of ConvNets. Every neuron in this layer connects to every neuron in the previous layer. Each neuron has a set of associated weights (as many neurons in the previous layer). The last fully-connected layer's output is typically a probability distribution over labels associated with the input data item. In Figure 2.3, the last fully-connected layer has four neurons. Together, these neurons provide an output that maps to a discrete probability distribution spread over four labels.

General Architecture of ConvNets. Typically a ConvNet architecture begins with one or more sequences of alternating convolutional and RELU layers, each followed by one or more pooling layers. This sequence of layers transforms the image from its original size to a smaller size and extracts features relevant to the classification task. Toward the end, there are one or more fully-connected layers. The last fully-connected layer holds the scores for every label in the classification task.

Training and Deploying ConvNets.. Convolutional neural networks can be trained in an end-to-end manner in the same way as fully-connected neural networks using gradient descent as described in Algorithm 2.1.

2.5 Deep Neural Network Ensembles

Ensemble learning is a prevalent approach to improve the quality of machine learning pipelines. The central idea is to train multiple diverse models to perform the same task and then combine their results during inference (Dietterich 2000). For instance, random decision forest, a popular ensemble learning algorithm, trains multiple (possibly hundreds) of individual decision tree models using the same training data. During deployment, the output from each of the decision trees is aggregated to provide the final output. Over the years, researchers and practitioners have used ensembles of virtually all machine learning algorithms to scale the capacity and accuracy of their machine learning pipelines (Drucker et al. 1993; Dietterich 2000; Granitto et al. 2005; Xu et al. 2014; Lee et al. 2015a; Russakovsky et al. 2015; Huggins et al. 2016).

Ensembles typically provide higher generalization accuracy than any of the individual models they contain. Various formal justifications have been provided to explain this. First, ensembles enrich the space of hypotheses considered and are representationally richer (Domingos 1999). Second, when these ensembles are combined, using some form of averaging, the models' variation, which is an artifact of stochastic training and the non-convex objective function, is reduced (Dietterich 2000).

Ensembles of deep neural networks are also increasingly used to scale the representational power of deep learning pipelines and improve their generalization accuracy (Huang

et al. 2017a; Wasay et al. 2020). They are widely used in online competitions such as the ImageNet challenge and COCO and applications with high accuracy requirements (Russakovsky et al. 2015; Huang et al. 2017a).

2.5.1 Training Deep Neural Network Ensembles.

There are two baseline approaches to train deep neural network ensembles depending on whether we use the entire data set or a different sample of the data set to train every network in the ensemble. The most prevalent method to train ensembles of neural networks is full data training. Here, we train all ensemble networks independently using the entire data set. This yields highly-accurate ensembles but also requires significant amount of training time. The second method, i.e., bagging or bootstrap aggregation, reduces the training time but comes at the cost of decrease in accuracy. Here, we train the network on a bagged set sampled from the training data. Given n training examples, we sample n times from it with replacement to create the bagged set. Effectively, this produces a training set that has less unique data items when compared with the full training data set.

For both of these methods – full data training and bagging – the individual network training proceeds as usual (using Algorithm 2.1). Formally, this yields k neural networks with k different functional mappings $\{f_1(\cdot, \boldsymbol{\theta}_1), ..., f_k(\cdot, \boldsymbol{\theta}_k)\}$.

2.5.2 Deploying Deep Neural Network Ensembles

During deployment, to classify a data item, we pass it through every network in the ensemble. This step produces a set of outputs, one for every network in the ensemble. These outputs are then combined to produce the final output.

Formally, any data item $\boldsymbol{x}^{(i)}$ is passed through each of the ensemble networks, producing k predictions $\{\hat{\boldsymbol{y}}_1^{(i)}, ..., \hat{\boldsymbol{y}}_k^{(i)}\}$. These k predictions are then combined together through an aggregation function (Ju et al. 2017).

Ensemble averaging and voting are two of the most commonly used aggregation functions. In ensemble averaging, the individual network outputs ($\{\hat{\boldsymbol{y}}_1^{(i)}, ..., \hat{\boldsymbol{y}}_k^{(i)}\}$) are aver-

aged to produce $\hat{\boldsymbol{y}}_e^{(i)}$, the final ensemble output for the i-th test data item.

$$\hat{\boldsymbol{y}}_e^{(i)} = \sum_{j=1}^{k} \hat{\boldsymbol{y}}_j^{(i)} / k$$

In voting, on the other hand, we look at the prediction of every neural network in the ensemble and output the prediction produced by a majority of the ensemble networks breaking ties randomly.

$$\hat{\boldsymbol{y}}_e^{(i)}[j] = \begin{cases} 1, & \text{if } j = mode(\arg\max(\hat{\boldsymbol{y}}_1^{(i)}), ..., \arg\max(\hat{\boldsymbol{y}}_k^{(i)})) \\ 0, & \text{otherwise} \end{cases}$$

2.5.3 Deep Neural Network Ensembles in Practice

Various applications increasingly use ensembles of multiple neural networks to scale the representational power of their deep learning pipelines. For example, deep neural network ensembles predict relationships between chemical structure and reactivity (Agrafiotis et al. 2002), segment complex images with multiple objects (Ju et al. 2017), and are used in zero-shot as well as multiple-choice learning (Guzman-Rivera et al. 2014; Ye and Guo 2017). Further, several winners and top performers on the ImageNet challenge are ensembles of neural networks (Lee et al. 2015a; Russakovsky et al. 2015). Ensembles function as collections of experts and have been shown, both theoretically and empirically, to improve generalization accuracy (Drucker et al. 1993; Dietterich 2000; Granitto et al. 2005; Huggins et al. 2016; Ju et al. 2017; Lee et al. 2015a; Russakovsky et al. 2015; Xu et al. 2014). For instance, by combining several image classification networks on the CIFAR-10, CIFAR-100, and SVHN data sets, ensembles can reduce the misclassification rate by up to 20 percent, e.g., from 6 percent to 4.5 percent for ensembles of ResNets on CIFAR-10 (Huang et al. 2017a; Ju et al. 2017).

While ensembles improve the quality of machine learning pipelines, they can also increase the design and maintenance cost as we need to first design and then maintain multiple models. Data scientists and machine learning practitioners use various strate-

gies to address these two challenges. To address the challenge of design cost, ensembles in practice contain neural networks that are architecturally homogenous, i.e., they replicate a single network design k times (Huang et al. 2017a). This replication eliminates the cost of ensemble design but results in some decrease in the diversity of the models. Similarly, there are practical approaches to reduce the cost of maintaining multiple models. The function learned from large ensembles can be transferred to smaller ensembles or even a single network while approximately preserving accuracy. In this way, ensembles of neural networks help learn a good function during training which is then transferred quickly to a more compact and manageable model (Hinton et al. 2015).

2.6 Transferring Knowledge between Neural Network Models

Transferring knowledge from one neural network to another neural network is a widely-used technique to increase or decrease a pre-trained model's size while retaining what the model has already learned. For instance, when datasets become more complex, machine learning practitioners might increase the effective complexity of the model and use knowledge transfer techniques to ensure that the enlarged model contains similar knowledge as the already-trained or deployed model. In addition to this, during transfer learning, data scientists may be interested in initializing a new neural network structure by using the knowledge from a pre-trained model from one task and then further train it for a new task. Knowledge transfer is used to accelerate network architecture search (Weill et al. 2019; Gordon et al. 2018), continuously expand neural networks to incorporate new data (Mitchell et al. 2018), and enable training of deeper neural networks (Romero et al. 2015; Simonyan and Zisserman 2015).

There are various techniques to transfer knowledge between two neural networks. They can be classified based on whether the knowledge transfer happens through further training or network transformation.

2.6.1 Knowledge Transfer Through Training

In the first class of approaches, the knowledge from a teacher network is transferred to a student network by training it (on possibly a transfer set) to mimic the teacher network (Hinton et al. 2015). This approach, known as knowledge distillation, trains the student network using the teacher network's probability distribution as the target. Suppose the labels corresponding to the transfer set are known. In that case, the loss function is a weighted sum of the empirical loss (i.e., the loss on the transfer set labels) and the target loss (i.e., the loss on the probability distribution produced by the teacher neural network model).

Formally, let $\boldsymbol{z}_t^{(l)}$ be the vector of outputs produced by the last layer of the teacher neural network on the l-th data item from the transfer set. Let $\boldsymbol{z}_s^{(l)}$ be the vector of outputs produced by the last layer of the student neural network. The total number of items in the transfer set are m^*. Then, the distillation loss J_D is defined as follows:

$$J_D = \sum_{l=1}^{m^*} \sum_{i=1}^{k} -p(\boldsymbol{z}_{ti}^{(l)}, T) \cdot \log(p(\boldsymbol{z}_{si}^{(l)}, T))$$

In this equation, p is the softmax probability scaled by a temperature factor T. This temperature factor controls how sharp or spread out the probability distribution is.

$$p(\boldsymbol{z}_i, T) = \frac{\exp(\boldsymbol{z}_i/T)}{\sum_{j=1}^{k} \exp(\boldsymbol{z}_j/T)}$$

The overall loss, is a weighted combination of the distillation loss and the empirical loss J_E of the student network.

$$J = \alpha \cdot J_D + (1 - \alpha) \cdot J_E$$

This loss is differentiable with respect to the student network parameters and training proceeds as usual using gradient descent as described in Algorithm 2.1.

Knowledge Distillation is a versatile technique and can transfer knowledge from any

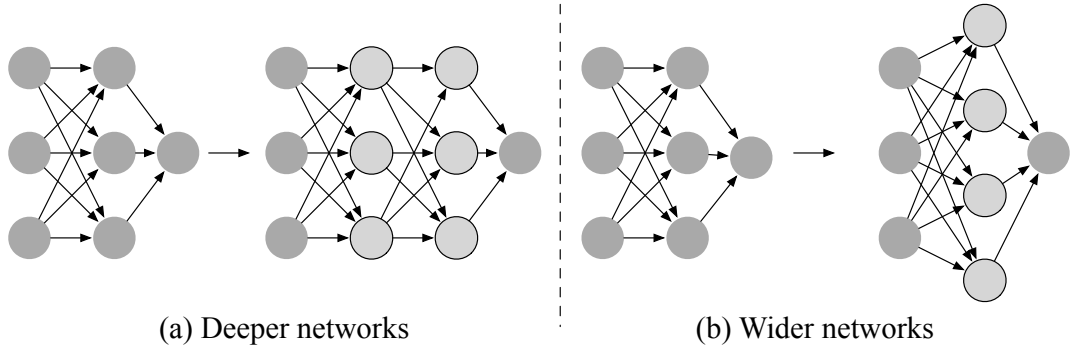


Figure 2.5: Function preserving transformations can be used to increase the depth and width of a given network, while preserving its function.

network to any other network. In past research, for instance, Knowledge Distillation has been used to incrementally train deep learning models and capture the knowledge from a large ensemble model into a smaller single model (Romero et al. 2015; Simonyan and Zisserman 2015). Knowledge distillation, however, requires extra training, and the knowledge is transferred only in an approximate way to the student network.

2.6.2 Knowledge Transfer Through Network Transformation

The second class of techniques to transfer knowledge does not require any extra training and ensures that the exact function gets transferred from the teacher to the student neural network (Chen et al. 2016). However, this approach lacks versatility and can only be used to grow the network's size, i.e., the student network needs to have the same or higher width, depth, and convolutional filter sizes compared to the teacher network. Overall, this approach, known as function-preserving transformation, expands the network by adding new parameters (increasing its width or depth) and ensuring that the newly-added parameters cause no change to the network's function.

We discuss three classes of function-preserving transformations as shown in Figure 2.5: (i) Net2DeeperNet that increases the depth of the network (Figure 2.5(a)), (ii) Net2WiderNet that widens the network layers (Figure 2.5(b)), and (iii) Net2WiderFilters that grows the size of convolutional filters (Figure 2.5(c)).

Net2DeeperNet. This transformation increases the depth of the network by adding a new layer to the network. It initializes the weights of the newly-added layer with identity matrices such that the input from the previous layer is passed unchanged to the next layer.

Specifically, this transformation replaces a layer $\boldsymbol{h}^{(i)} = \phi(\boldsymbol{h}^{(i-1)}, \boldsymbol{W}^{(i)})$ with two layers $\boldsymbol{h}^{(i)} = \phi(\boldsymbol{U}^{(i)}, \phi(\boldsymbol{h}^{(i-1)}, \boldsymbol{W}^{(i)}))$. The parameters of the new layer $U^{(i)}$ are initialized with identity and this transformation is only applicable when $\phi(\boldsymbol{I}, \boldsymbol{v}) = \boldsymbol{v}$ for any vector $\boldsymbol{v}$. This holds for many widely-used layers such as convolutional layers, fully-connected layers, and ReLU layers. While this transformation cannot be exactly applied to layers having sigmoid or tanh activation functions, it has been shown empirically that the resultant transformation still provides some degree of function preservation.

Net2WiderNet. This transformation replaces an existing layer with a wider layer, i.e., a layer with more neurons. In the case of convolutional neural networks, this means a layer with more convolutional filters.

Suppose we have two layers i and $i+1$ such that layer i has m inputs and n outputs, whereas layer $i+1$ has n inputs and p outputs. The associated weights $\boldsymbol{W}^{(i)}$ and $\boldsymbol{W}^{(i+1)}$ belong to $\mathbb{R}^{m\times n}$ and $\mathbb{R}^{n\times p}$ respectively. Net2WiderNet allows us to replace layer i with a layer that has $q > n$ outputs. To do this, first, we replace $\boldsymbol{W}^{(i)}$ and $\boldsymbol{W}^{(i+1)}$ by $\boldsymbol{U}^{(i)}$ and $\boldsymbol{U}^{(i+1)}$ respectively. $\boldsymbol{U}^{(i)}$ is constructed such that the first n columns are exactly the same as $\boldsymbol{W}^{(i)}$ and the remaining columns ($n+1$ through q) are chosen randomly from the first n columns. To neutralize the effect of this replication, weights in $\boldsymbol{W}^{(i+1)}$ corresponding to a replicated column in $\boldsymbol{W}^{(i)}$ are scaled down by its replication factor. More specifically the following function enables this transformation:

$$\boldsymbol{U}^{(i)}_{k,j} = \boldsymbol{W}^{(i)}_{k,g(j)}, \qquad \boldsymbol{U}^{(i+1)}_{j,h} = \frac{1}{|\{x|g(x) = g(j)\}|}\boldsymbol{W}^{(i+1)}_{g(j),h}$$

$$\text{Where:} \quad g(j) = \begin{cases} j & j \leq n \\ \text{random sample from } \{1,2, \ldots \text{ n}\} & j > n \end{cases}$$

Further Training Transformed Networks. Once we apply one or more of these transformations to a pre-trained network, it has the same function as before but more parameters we can further train. We add random noise to perturb these new parameters to break symmetry and avoid zero-initialized weights. The network is then trained as before on the training data set.

3

Related Work

Our work on Computation-Cautions Machine Learning Systems is related to various research directions from data systems and machine learning research communities. In particular, we position our work with respect to systems and techniques that improve various stages of machine learning pipelines, specifically those that (i) improve data exploration, (ii) enable better understanding of deep learning model design, and (iii) accelerate training and deployment of deep learning models.

3.1 Efficient Data Exploration

Data exploration is the first step in any data science pipeline, where a data scientist is interested in understanding various properties of the data set. During exploration, a data scientist is looking for interesting patterns in massive data sets that are not known a priori (Idreos 2013; Wasay et al. 2015). This stage presents unique challenges of scalability, efficiency, and usability. We discuss research directions that tackle these challenges and explain how our work in Data Canopy complements and advances state-of-the-art data exploration techniques.

3.1.1 Data Exploration and Statistics

Statistics play an important role in data exploration as they summarize data and help uncover the relationship between different attributes in the data set. As such, systems used for exploratory analysis, including data systems, support the computation of statistics. These systems also use statistics to inform various internal optimization processes. As such, existing research directions design ways to utilize statistics and improve and enhance their computation.

Statistics in Data Systems

Statistics are widely used within data systems to analyze data and tune various internal components of the database system: Data systems provide support to compute different statistics in the form of aggregate operations such as AVG and CORR (Zhao et al. 1997). Also, query optimizers estimate query cardinality by using histogram statistics (Chaudhuri and Narasayya 1998). Recent approaches employ statistics for data integration (Halevy 2004; Dalvi and Suciu 2005), time series analysis (Sathe and Aberer 2013; Zhu and Shasha 2002), and in-database learning (Schleich et al. 2016).

Data Canopy vs. Statistics in Data System. Despite the widespread use of statistics in data systems, a framework to synthesize and reuse various statistical measures during exploratory statistical analysis does not exist. Data Canopy introduces such a framework, which replaces ad hoc calculation of statistics and brings opportunities to synthesize statistics from basic aggregates efficiently; compute and cache these basic aggregates ahead of time, and employ them to accelerate exploratory statistical analysis. Statistics in Data Canopy, primarily computed for exploratory analysis, can also be used within the data system for other tasks such as query optimization, data integration, and data discovery and mining.

Fast Computation of Statistics

The widespread use of statistics in various systems across the board has led to research on calculating fast statistics on large data sets. One set of research directions reduces the amount of data touched to compute statistics while providing accuracy guarantees: Robust sampling techniques are applied to trade accuracy for performance (Gibbons et al. 1997; Chaudhuri et al. 1998, 2004; Cormode and Muthukrishnan 2005; Wu et al. 2010) and techniques based on discrete Fourier transform approximate all-pair correlations for time series (Mueen et al. 2010). Other research directions present solutions to computing statistics at scale in distributed settings: Cumulon is an end-to-end system, which optimizes the cost of calculating statistics on the cloud (Huang et al. 2013). Similarly, some research directions optimize the calculation of various statistical measures by correctly partitioning data in distributed settings (Cormode and Muthukrishnan 2005; Alvanaki and Michel 2014).

Data Canopy vs. Fast Statistics. All these approaches innovate on how statistics are computed. Therefore, these approaches are compatible with Data Canopy: Data Canopy can adopt one or even multiple approaches for computing basic aggregates. For example, in distributed settings, Data Canopy can incorporate the aforementioned partitioning techniques to ensure that relevant data is stored at local nodes. Similarly, the computation of statistics in Data Canopy can be done approximately. The primary advantage of combining Data Canopy with these approaches is that Data Canopy synthesizes statistics from basic aggregates and reuses these basic aggregates. In the presence of workloads exhibiting high locality and repetition, this synthesis significantly reduces data movement.

3.1.2 Novel Data Exploration Paradigms

Data Exploration has received a lot of research interest within the data systems community (Idreos et al. 2015; Wasay et al. 2015).

First, there are *exploratory interfaces* that steer data scientists through the data space by providing both insights and further queries: Recent approaches discover relevant

data objects based on relevance-feedback (Dimitriadou et al. 2014) or by performing a variety of faceted search (Drosou and Pitoura 2013); Query recommendation systems help data scientists ask relevant questions based on the data set and their past interests (Yu et al. 2010; Abouzied et al. 2012; Shen et al. 2014).

Second, there is research to enable *visual analytics* that reduce the cognitive effort of data exploration by augmenting data systems with visual and gestural interfaces: Various approaches enable data scientists to visually browse data sets (Stonebraker and Kalash 1982; Stolte et al. 2002; Wu et al. 2014); Recommendation systems automatically select an appropriate visualization given a data set (Key et al. 2012); DbTouch (Liarou and Idreos 2014) and GestureDB (Nandi 2013) develop database kernels and languages that fingertips can control; Visualization languages enable declarative definition of data visualizations (Hanrahan 2006).

Finally, *approximate query processing* provides estimated answers to exploratory queries in orders of magnitude less time by touching a fraction of base data. There are also proposals to use samples of the data set to answer queries while satisfying a user-defined accuracy (Gibbons et al. 1998; Sidirourgos et al. 2011; Agarwal et al. 2013).

Data Canopy and Novel Data Exploration Paradigms. Data Canopy as a framework for exploratory statistical analysis is complementary to all efforts above. None of the works described above are about making the process of computing statistics more interactive. Data Canopy can help make any process that contains iterative computation of statistics more interactive. Similarly, recommendation systems can use various descriptive and dependence statistics for faster and more informed recommendations. Data Canopy can also benefit from many of these research directions in the general field of data exploration. For instance, Data Canopy can use sampling and approximation techniques to create a smart cache with approximate guarantees.

3.1.3 Data Cubes

Data cubes, widely applied in mining data warehouses, store data aggregated across multiple dimensions (Gray et al. 1997; Mumick et al. 1997). Operators like roll-up, slice, dice, drill-down, and pivot allow data scientists to summarize or further resolve

information along any particular dimension in the data cube. Various techniques to improve data cube performance have been studied: Sampling and other approximation techniques are used to reduce both the time required to construct the data cube and answer queries from it (Barbara and Sullivan 1997; Li et al. 2008; Xie et al. 2016). Some approaches only partially materialize data cubes (Dyreson 1996; Xin et al. 2003; Feng et al. 2004), whereas others present strategies to build them adaptively (Beyer and Ramakrishnan 1999), and in parallel settings (Chen et al. 2006). One line of work proposes a simplified and flexible version of the data cube concept in the form of small aggregates (Moerkotte 1998). Furthermore, recent research designs data cubes for exploratory data analysis: Some research directions visualize aggregates stored in data cubes (Kahng et al. 2016), others use them for ranking (Wu et al. 2008) as well as for interactive exploration (Sarawagi and Sathe 2000).

Data Canopy vs. Data Cubes. Data cubes do not support a wide range of statistical measures. Specifically, they have no support for multivariate statistics such as correlation or covariance. Also, data cubes come with a high preprocessing and memory cost resulting from calculating and storing aggregates grouped by multiple dimensions. In contrast, Data Canopy is lightweight and can reuse and synthesize an extendible set of statistics using a relatively small set of basic aggregates. Furthermore, slices obtained from data cubes in OLAP settings can be explored using Data Canopy. Once data scientists have developed an understanding of the data set, they can construct more complicated OLAP structures or run more detailed analytics on features and subsets of data that they have identified to be of interest. This approach is more efficient compared to building heavy OLAP structures upfront for exploratory statistical analysis.

3.1.4 Query Caching and Materialized Views

Query result caching enables database systems to reuse results of past queries to speed up future queries (Hellerstein and Naughton 1996). Most relevant to Data Canopy are approaches that enable reuse across different ranges by breaking down queries and caching query results (Keller and Basu 1996; Deshpande et al. 1998). More recently, different approaches prefetch both data and query results to accelerate the process of data exploration. Forecache breaks the data down into regions called tiles and prefetches

them based on a data scientist's exploration signature (Battle et al. 2016). Similar caching and prefetching strategies have also been proposed for the process of data visualization (Liu et al. 2013).

All relational databases, such as MonetDB and Postgres, provide support for storing results of previously executed queries to speed up future queries in the form of materialized views (Chirkova and Yang 2011). Recent research explore various methods to figure out what materialized views are selected: There are proposals to use stochastic optimization as well as evolutional algorithms to decide on what views to materialize automatically (Kumar and Kumar 2018; Gosain and Sachdeva 2019; Sohrabi and Azgomi 2019). In addition to this, there is research on how to efficiently maintain and store the metadata resulting from materialized views, including incremental computation, proper partitioning in a distributed setting, and extending to key-value stores (Du et al. 2017; Ahmed et al. 2020; Adler 2020). Finally, there is a recent proposal to replace materialized views with learned models (Zou 2021).

Data Canopy vs. Query Caching. Data Canopy draws inspiration from these approaches and takes a step further: In addition to decomposing ranges, Data Canopy decomposes statistical measures into a set of basic aggregates that can be reused between them. As such, Data Canopy can synthesize descriptive and dependence statistics directly from this library of basic aggregates. Additionally, Data Canopy advances this research direction by providing a smart cache framework that can compute and maintain a library of basic aggregates that can be used as building blocks for various statistical measures and machine learning algorithms.

Data Canopy and Materialized Views. It is possible to use materialized views to implement Data Canopy in relational database systems. In particular, we can store basic aggregates in the form of materialized views, i.e., aggregates such as COUNT, SUM, etc., over the base data. To fully support Data Canopy in relational database systems through materialized views, we will need to implement: (i) a function that intercepts user queries and decomposes statistics into corresponding basic aggregates and maps them to materialized views, (ii) a function that executes the creation of materialized views that we have not yet created, (iii) a meta-data table that stores what materialized views we have already created, and (iv) a function that utilizes information from the

meta-data table to synthesize results (from existing materialized views) instead of going back repeatedly to the base data. We can further optimize the meta-data table to take the form of segment trees, i.e., the range composable data structures that Data Canopy uses to store and retrieve basic aggregates in logarithmic time.

Another method to incorporate Data Canopy into relational database systems through materialized views would be to reimplement materialized views so that they store statistical queries in the form of basic aggregates, i.e., instead of storing a single number corresponding to a statistic on the data set, the corresponding basic aggregates are stored and maintained. Overall, Data Canopy brings an opportunity to reuse work across different materialized views to relational database systems.

3.1.5 Incremental Stream Processing

Similarly, in streaming scenarios, incremental query processing decomposes data streams into smaller chunks and runs queries on these chunks: Window-based approaches partition data and queries such that future windows can make use of past computation (Chandramouli et al. 2014; Chandrasekaran et al. 2003; Ghanem et al. 2007; Liarou et al. 2013). Specific approaches present strategies to incrementally monitor time-series data (Zhu and Shasha 2002) and update materialized views (Griffin and Libkin 1995).

Data Canopy and Stream Processing. Data Canopy is inspired by these approaches and is readily applicable in streaming settings as it can be constructed in a single pass over the data set. When processing massive streams with limited memory, Data Canopy can function as a synopsis for answering a configurable set of statistical queries for exploratory statistical analysis. This synopsis is constructed and updated incrementally.

3.2 Understanding Deep Learning Model Design

Model design is the next stage of data science and machine learning pipelines. Researchers have extensively studied deep learning model design from various angles. Recent work conducts large-scale experiments to derive guidelines to enable better model design. In the absence of a robust theoretical understanding of the design space, model

designers can use these guidelines to bootstrap their design process. Here, we review research directions connected to Deep Collider, our framework that enables a principled and holistic understanding of deep learning ensembles' model design space. We position Deep Collider against existing empirical research on designing single network models, ensembles of models, and the design space between these two alternatives.

3.2.1 Single Network Design Space

Past empirical studies have progressively disentangled the contribution of three design choices – depth, width, and convolution operators – to the accuracy of convolutional networks (Ba and Caruana 2014; Eigen et al. 2013; Novak et al. 2018; Urban et al. 2016; Zhao et al. 2018). The methodology is to synthesize neural network architectures that can isolate the effect of these design choices and, then, conduct large-scale experiments over various types of architectures, data sets, and hyperparameter configurations. This methodology has lead to several insights on how to design single neural network models: Researchers have shown that depth is crucial as it increases non-linearity in the network (Ba and Caruana 2014). Similarly, research establishes that the convolution operator is essential to achieving good generalization accuracy (Urban et al. 2016). The number of filters per layer, on the other hand, is shown to be of little consequence as long as it is above a threshold (Eigen et al. 2013).

Single Network Design Space and Deep Collider. Our study pushes this line of research forward in two significant ways: First, we consider the dimension of ensembling and compare it with a single deep network under a parameter budget. This analysis provides model designers with a way to reason about using an ensemble of deep neural networks in their pipeline. Second, in addition to the metric of generalization accuracy, we also include training time, inference time, and memory usage. These metrics are of increasing importance as deep learning models are trained and deployed in heterogenous settings with varying amounts of computational resources.

3.2.2 Ensemble Design Space

Closely related are recent studies that compare the generalization accuracy of ensembles of neural networks with single networks (Russakovsky et al. 2015; Lee et al. 2015a; Ju et al. 2017). These studies, however, compare the generalization accuracy of single deep networks with ensembles, where each network in the ensemble has the same size as the single deep network, i.e., the number of parameters in the ensemble are significantly higher as compared to the single network. For instance, one such study compares an ensemble of four ResNets-110 with a single base model (Ju et al. 2017). The consensus from these studies is to use ensembles when the goal is to achieve high accuracy without much regard to training cost, inference time, and memory usage, e.g., for competitions such as COCO and ImageNet.

There are recent studies that make this comparison between ensembles and single network models under a parameter budget. The major conclusion is that beyond a specific budget, ensembles can provide better accuracy than single networks (Chirkova et al. 2020; Kondratyuk et al. 2020). This work, however, considers only the metric of generalization accuracy and explores a minimal design space – two different classes of convolutional architectures with a single depth.

Ensemble Design Space and Deep Collider. Our study, in contrast, compares single networks to ensembles under a fixed parameter budget, which is crucial for a fair comparison as the training and deployment cost depends on the number of parameters. Then, we investigate how this apples-to-apples comparison evolves as we isolate and vary the number of networks, the depth per network, and the number of filters per network in the ensemble. As such, our work significantly extends the existing line of inquiry, not just answering the question of how a single network compares with a K-network ensemble (with K times as many parameters), but also addressing whether an ensemble with the same number of total parameters as a single model can be better and when.

Compared with recent studies that make this comparison under a parameter budget, Deep Collider brings a distinctive advantage: We conduct this analysis while considering a holistic set of metrics that include resource-related metrics such as training time, inference cost, and memory usage. All these metrics are critical for practical applications

(Sze et al. 2017a). Furthermore, to provide reliable guidance to a model designer, a robust assessment must consider a wide range of architectures and model sizes with various depth and width configurations. This consideration is critical, primarily because varying just the width of convolutional networks in isolation, as done by recent studies (Chirkova et al. 2020; Kondratyuk et al. 2020), is known to be far less effective to improve accuracy (Eigen et al. 2013; Ba and Caruana 2014).

3.2.3 Theoretical Frameworks

Researchers are gradually developing theoretical frameworks to understand and explain various deep learning models (Arora et al. 2018; Eldan and Shamir 2015; Lu et al. 2017; Mhaskar et al. 2016; Telgarsky 2016). Theoretical frameworks in deep learning are motivated by the need to explain and generalize empirical observations. For instance, after empirical research established that depth is vital to the generalization accuracy of neural networks, a theoretical framework was recently developed to explain the power of depth (Eigen et al. 2013; Eldan and Shamir 2015).

Theoretical Frameworks and Deep Collider. Our study uncovers various observations about the behavior of training a given parameter budget in a single deep network and an ensemble of multiple networks. It builds the basis for enabling future theoretical frameworks to capture the relationship between ensembles and deep networks.

3.3 Efficient Deep Learning

As deep learning pipelines become more and more widespread, an ecosystem of research has emerged to improve their efficiency targeting the training of individual neural networks. Various algorithmic and system techniques target fundamental bottlenecks in the training process such as gradient descent and matrix operations (Niu et al. 2011; Brown et al. 2016; Li et al. 2019b,a): Various research directions improve parallelism by introducing asynchrony in updating weights during backpropagation. Other research directions propose adaptively tuning training hyperparameters (Bottou et al. 2016). In addition to these, systems researchers develop techniques to reduce data movement and

memory overhead. Approaches include using low-precision to reduce the amount of data touched (De Sa and Feldman) and efficient encoding of activations during training (Jain et al. 2018). Finally, specialized hardware is being developed both in industry and research labs to improve performance, parallelism, and energy consumption of the training process (De Sa and Feldman; Prabhakar et al. 2016; Wang et al. 2020).

MotherNets vs. Efficient Deep Network Training. All techniques to improve upon training efficiency of individual neural networks are orthogonal to MotherNets and in fact directly compatible. This is because MotherNets does not make any changes to the core components of the training process of each individual network i.e., forward pass and back propagation. In our experiments, we do utilize some of the widely applied training optimizations such as batch-normalization and early-stopping. The advantage that MotherNets bring on top of these approaches is that we can now reduce the total number of epochs that are required to train an ensemble. This is because a MotherNet will train for the structural similarity present in the ensemble once.

3.4 Fast Ensemble Training and Deployment

In addition to methods to improve general training of deep neural networks, techniques have also been introduced to accelerate neural network ensemble training. Fast ensemble training techniques fall under three different categories: (i) Techniques that generate ensembles by training a single network, (ii) Techniques that implicitly or explicitly share parameters between training of various ensemble networks, and (iii) Techniques that use knowledge distillation to bootstrap the ensemble training process.

3.4.1 Generating Ensembles From Single Networks

The first class of techniques we discuss generates a target neural network ensemble by training just a single neural network instead of training every network in the ensemble. Overall, these approaches can train an ensemble in an amount of time comparable to the time it takes to train a single network model. However, this improvement in training time comes at the cost of reduced accuracy compared to an approach that trains

every network in the ensemble separately. This class of techniques includes Snapshot Ensembles and Fast Geometric Ensembles.

Snapshot Ensembles. Snapshot Ensembles train a single network and use its parameters at k different points of the training process to instantiate k networks that will form the target ensemble (Huang et al. 2017a). Snapshot Ensembles vary the single network's learning rate cyclically, forcing it to visit several local minima. In particular, Snapshot Ensembles use cyclical cosine annealing to vary the learning rate: Every cycle starts with a large learning rate that is annealed to a lower learning rate. The large learning rate dislodges the neural network model from its current local minima, whereas the lower learning rate enables it to visit a well-behaved new local minima. At every local minima, Snapshot Ensembles save a copy of the single network's parameters. These neural network copies function as different members of the generated ensemble.

Fast Geometric Ensembles. Fast Geometric Ensembles, closely related to Snapshot Ensembles, is another approach to generate ensembles from the learning trajectory of a single model (Garipov et al. 2018). Fast Geometric Ensembles also train a single neural network architecture and saves the network's parameters at various training trajectory checkpoints. Instead of using cosine annealing, Fast Geometric Ensembles use a cyclical geometric learning rate to explore various local minima in the neural network's loss surface, saving a copy of the network parameters at every minima.

MotherNets vs. Fast Ensemble Generation. We draw motivation from this line of work; however, in contrast to these approaches that generate ensembles where members have a monolithic architecture, MotherNets accelerate training of large ensembles with diverse neural network architectures enabling structural diversity. Furthermore, these approaches can all be used in conjunction with MotherNets to generate additional ensembles from different neural network architectures.

3.4.2 Training Ensembles by Parameter sharing

We discuss the next set of techniques to reduce ensemble training and deployment time by sharing parameters between different networks in the ensemble. This sharing can

happen explicitly, as is the case with TreeNets, or, in the case of Dropout and Pseudo-ensembles, this sharing can happen implicitly.

Explicit Sharing: TreeNets. Given a neural network ensemble, TreeNets trains the parameters of the first few (two to four) layers in a shared manner (Lee et al. 2015b). Specifically, TreeNets combine the initial few layers of various networks in the given ensemble to create a network that branches into k sub-networks (where k is the number of networks in the ensemble) after the first few shared layers. Effectively every sub-network functions as a separate member of the target ensemble. This network with multiple branches, called the TreeNet, is then trained in an end-to-end manner minimizing a composite loss function that consists of (i) the canonical empirical loss and (ii) a regularization term that induces diversity between different sub-networks.

MotherNets vs. TreeNets. MotherNets is different from TreeNets in two regards. First, MotherNets does not restrict the size and architecture of models in the ensemble, i.e., there is no requirement for initial layers to have the same structure (as is the case with TreeNets). In other words, MotherNets can train an ensemble with arbitrary size and containing networks with diverse architectures. Second, MotherNets train the ensemble in two phases that explicitly first lower bias and then create diversity.

Implicit Sharing. As opposed to the explicit sharing of parameters in the initial layers of TreeNets, various existing approaches create many networks with shared weights during training and implicitly ensemble them during inference. These techniques proceed by zeroing out a random subset – individual nodes, connections, and complete layers – of a network during every round of mini-batch training. During inference, these approaches scale every part of the trained network by its probability of surviving during training (Wan et al. 2013; Srivastava et al. 2014; Singh et al. 2016; Huang et al. 2016).

MotherNets vs. Pseudo-Ensembles. MotherNets capture the structural similarity in an ensemble, where members have different and explicitly defined neural network architectures in the form of MotherNets. After training for this similarity, we transfer well-trained parameters to all ensemble networks that are further trained. Overall, this enables us to combine well-known architectures within an ensemble effectively.

	Fast training	High accuracy	Diverse architecture	Large ensemble size
Full data	×	✓	✓	×
Bagging	∼	×	✓	×
Knowledge Distillation	∼	×	✓	×
TreeNets	∼	∼	×	×
Snapshot Ensembles	✓	✓	×	×
Fast Geometric Ensembles	✓	✓	×	×
MotherNets	✓	✓	✓	✓

Table 3.1: Existing approaches to train ensembles of deep neural networks are limited in speed, accuracy, diversity, and size. MotherNets overcomes all these shortcomings.

Furthermore, implicit ensemble techniques, for instance, Dropout and Swapout, can be used as optimizations to further improve upon the generalization accuracy of individual networks trained through MotherNets (Srivastava et al. 2014; Singh et al. 2016).

3.4.3 Training Ensembles by Knowledge Distillation

Knowledge Distillation provides a middle ground between separate training and ensemble generation approaches (Hinton et al. 2015). Knowledge Distillation trains an ensemble in two steps: First, it trains a large *generalist* network and then distills its knowledge to an ensemble of small *specialist* networks that may have different architectures (by training them to mimic the probabilities produced by the larger network) (Hinton et al. 2015; Li and Hoiem 2017).

MotherNets vs. Knowledge Distillation. The major drawback of using Knowledge Distillation to train ensembles is that distilling knowledge still takes around 70 percent of the time needed to train every neural network from scratch. Even then, the ensemble networks are still closely tied to the same large network that they are distilled from. The result is significantly lower accuracy and diversity when compared to ensembles where every network is trained individually (Hinton et al. 2015; Li and Hoiem 2017). MotherNets, on the other hand, avoids this training cost by hatching the ensemble networks through rapid function-preserving transformations. Once the networks have been hatched, they are trained independently, allowing the hatched networks to explore

a different solution space than that of their MotherNet. Overall, this results in better accuracy and faster training as compared to training via Knowledge Distillation.

Table 3.1 provides an overall comparison between MotherNets and all other fast ensemble training approaches. All other approaches lack in one or more of the three dimensions of speed, accuracy, diversity, and scalability. MotherNets, on the other hand, is the first general purpose fast ensemble training approach that extends to ensembles of diverse neural network architectures while providing high accuracy and low training time. As such it enables training of large ensembles consisting of multiple single neural networks.

4

Data Canopy: Accelerating Exploratory Data Analysis

We now present Data Canopy in detail. Data Canopy enables data scientists to perform exploratory statistical analysis without having to repeatedly scan the entire base data. The main idea is that Data Canopy breaks statistics down to basic aggregates. It caches and manages a library of basic aggregates so that incoming queries may use it to synthesize different kinds of statistics. Data Canopy can compute the library of basic aggregates in a single offline pass over the data. For dynamic data exploration scenarios with little idle time, Data Canopy incrementally computes the library of basic aggregates during query processing.

4.1 Example: Query Processing in Data Canopy

Before we discuss the design of Data Canopy, we motivate and provide the core intuition with the help of an example. Consider the hourly temperature measurements collected by the National Centers for Environmental Information (NCEI) (nce 2016). On this data set we build an instance of Data Canopy that is configured to work with three

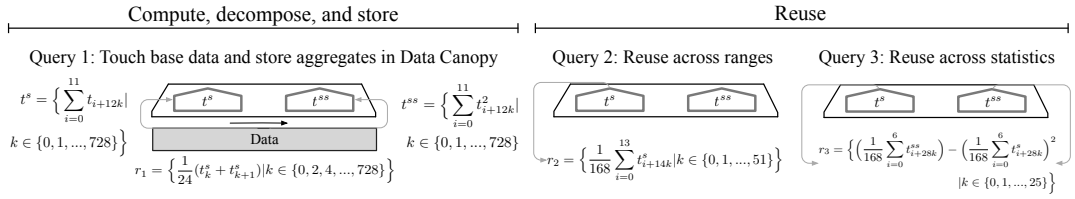


Figure 4.1: An example of queries that can reuse computation and data access through Data Canopy.

univariate statistics: mean, variance, and standard deviation. Figure 4.1 shows how Data Canopy processes a series of queries over this data set without having to always check the base data.

Query 1: The data scientist requests mean temperatures for each day. Data Canopy is initially empty i.e., there are no basic aggregates to utilize. For this query Data Canopy has to access base data and compute the daily mean temperatures (using 24 observations for each calculation). Data Canopy takes this opportunity to compute and store two types of basic aggregates: (1) basic aggregates that are immediately needed to synthesize statistics for the current query, and (2) basic aggregates that are not immediately needed, but can be computed from accessed data and then reused by other statistics. These basic aggregates are always maintained at a fixed granularity of a chunk. For ease of presentation, the chunk size is set to 12 in this example, i.e., one chunk corresponds to twelve hours (in practice Data Canopy autotunes the chunk size as we will discuss later on). The basic aggregates resulting from this query are shown under Query 1 in Figure 4.1. For every chunk of size 12, Data Canopy stores the set of sums (t^s), to be used for the current query, and the set of sums of squares (t^{ss}), that may be used by future queries (for example for standard deviation and variance).

Query 2: The data scientist requests mean temperatures for each week. This time the data scientist asks for the same statistic as requested in Query 1 but at a different granularity (weekly instead of daily). As shown under Query 2 in Figure 4.1, there is no need to access the base data again. Data Canopy already contains t^s, the sums of hourly temperatures for every 12 hours. It sums up 14 consecutive values of t^s to synthesize the result for each week.

Query 3: The data scientist requests variances in temperature for every two weeks. This time the data scientist asks for both a different statistical measure and at a different granularity (biweekly instead of weekly or daily). As shown under Query 3 in Figure 4.1, Data Canopy synthesizes statistics from basic aggregates, again, without accessing the base data. The variance of a set of observations x is given by Equation 4.1. Data Canopy thus uses t^s and t^{ss} to synthesize the result set r_3 for this query.

$$v_x = \Big(\frac{1}{N}\sum_{i=1}^{N} x_i^2\Big) - \Big(\frac{1}{N}\sum_{i=1}^{N} x_i\Big)^2$$

Other Queries. Similar to the above scenarios, once Data Canopy stores the set of sums t^s for every 12 hours, and the set of sums of squares t^{ss} for every 12 hours, it can reuse these basic aggregates in four different types of query scenarios:

i. *Across different data ranges*: daily mean of the first three days, daily mean of the last four days, etc.

ii. Across different data granularities: weekly mean, biweekly mean, etc.

iii. *Across different statistical measures*: daily standard deviation, daily variance, etc.

iv. *Across any combinations of i, ii, and iii*: weekly standard deviation, monthly variance, weekly range etc.

4.2 Design Concepts

We now describe the core design concepts in Data Canopy that enable the aforementioned degree of reuse.

Data and Query Range. We will use the concepts of data and query range throughout our discussion. We define a data range as a set of consecutive data items from a column or a set of columns. A query range is the data range over which a query requests statistical measures.

Statistics		Basic Aggregates				
Type	*Formula*	$\sum x$	$\sum x^2$	$\sum xy$	$\sum y^2$	$\sum y$
`avg`	$\frac{\sum x_i}{n}$	■				
`rms`	$\sqrt{\frac{1}{n} \cdot \sum x^2}$		■			
`var`	$\frac{\sum x_i^2 - n \cdot \texttt{avg}(x)^2}{n}$	■	■			
`std`	$\sqrt{\frac{\sum x_i^2 - n \cdot \texttt{avg}(x)^2}{n}}$	■	■			
`kur`	$\frac{1}{n} \sum (\frac{x_i - \texttt{avg}(x)}{\texttt{std}(x)})^4 - 3$	■	■			
`cov`	$\frac{\sum x_i \cdot y_i}{n} - \frac{\sum x_i \cdot \sum y_i}{n^2}$	■		■		■
`slr`	$\frac{\texttt{cov}(x,y)}{\texttt{var}(x)}, \texttt{avg}(x), \texttt{avg}(y)$	■	■	■		■
`corr`	$\frac{n \cdot \sum x_i \cdot y_i - \sum x_i \cdot \sum y_i}{\sqrt{n \cdot \sum x_i^2 - (\sum x_i)^2} \sqrt{n \cdot \sum y_i^2 - (\sum y_i)^2}}$	■	■	■	■	■

Table 4.1: Data Canopy synthesizes statistics from a library of basic aggregates.

Basic Aggregates. Data Canopy breaks statistical measures into basic primitives. We call those primitives basic aggregates. We define a basic aggregate over a data range as a value that is obtained by first performing a transformation τ on every data item in that data range and then combining the results using an aggregation function f. Formally, for a given data range X, (with elements x_i) a basic aggregate can be represented as $f(\{\tau(x_i)\})$. In our running example, sum of squares t^{ss} can be represented as $f(\{\tau(x_i)\})$ $= \sum_i x_i^2$, where $\tau(x_i) = x_i^2$ and f is the sum function.

The transformation τ can be any operation on an individual data item. However, the aggregation function f has to be commutative and associative i.e., we should be able to break down and combine basic aggregates between sub-ranges (partitions of the data range). Formally, for any partition $\{X_1, X_2, \ldots X_n\}$ of a data range X, the following should hold:

$$f(X) = f(\{f(X_1), f(X_2) \ldots f(X_n)\}) \tag{4.1}$$

For instance, this property is satisfied by min, max, count, sum, and product functions on any given data range, whereas the median function does not satisfy this property.

Decomposing Statistics. Data Canopy defines a statistic S over a data range X as a function F of different basic aggregates:

$$S(X) = F(\{f(\tau(\{x_i\}))\})$$

Figure 4.3 shows how statistic S (with function F) is mapped to two basic aggregates. The rationale behind representing statistics as a function of basic aggregates is twofold: First, various statistical measures share – and can reuse – basic aggregates. For instance mean, variance, and standard deviation all require the basic aggregate of sum over the target data. Second, a given basic aggregate over a certain data range (as a result of the property in Equation 4.1) can be further decomposed into sub-ranges. These sub-ranges can be combined together to synthesize that basic aggregate over any data range that contains those sub-ranges.

Table 4.1 shows how Data Canopy breaks down a set of widely used descriptive and dependence statistics into five basic aggregates. Effectively, Data Canopy is a smart cache. An alternative approach could be that we cache the result values of each individual statistic. However, we then lose the ability to reuse computation and data access between different statistics, despite clear overlaps. For instance, if instead of caching each of the basic aggregates corresponding to correlation, we cached just the final value, we will not be able to use that value to synthesize any of the other statistical measures mentioned in Table 4.1. Instead, we would have to access the data set again to compute the individual statistics.

In addition to the examples in Table 4.1, geometric mean ($\tau(x) = x, f(X) = \prod_i x_i$), harmonic mean ($\tau(x) = \frac{1}{x}, f(X) = \sum_i x_i$) and other descriptive and dependence statistics can be synthesized from basic aggregates. Over 90 percent of statistics supported by NumPy and SciPy (num 2013), and over 75 percent of statistics supported by Wolfram (Wol 2017) can be expressed in the aforementioned form i.e., they can be decomposed and expressed in terms of τ, f, and F.

Chunks. Data Canopy maintains basic aggregates at the granularity of a chunk – a logical partition of data that comprises of consecutive values from a data column. For every chunk, Data Canopy maintains a single value per basic aggregate type. In

our example of hourly temperature data, a chunk size of 12 implies that for every statistical measure that Data Canopy computes, it caches each of the resulting basic aggregates over every 12 data values. This concept of chunk is essential to how Data Canopy enables reuse – reducing repeated data access – between different queries during exploratory statistical analysis.

As a result of chunking, queries of any data range larger than the chunk size can be synthesized directly from basic aggregates. Even in cases when the query range does not exactly align with the chunks, Data Canopy only needs to scan at most the two chunks at the edges of the requested query range. In a similar fashion, queries having partial range overlaps with previously computed chunks can also reuse basic aggregates. Mapping this concept to our running example, weekly and yearly variances in temperature can be synthesized from daily aggregates. Also, a query that requests the mean temperature over the last three weeks of a month, can reuse overlapping basic aggregates corresponding to the first two weeks.

Overall. Data Canopy is able to reuse previously computed basic aggregates to synthesize a wide set of statistics. As a concrete example, by storing just two basic aggregates of sum and sum of squares over five chunks in ten columns (a total of 100 values), Data Canopy can reuse this information across queries that target 2^5 possible combinations of chunks and request for up to four statistical measures – mean, variance, root mean square, and standard deviation – over any of these ten columns.

4.3 Data Structure

Data Canopy uses a set of segment trees to store basic aggregates. Segment trees support efficient aggregate queries over a data range without the need to access individual data items (Saxe and Bentley 1979; De Berg et al. 2000). This property is satisfied by storing, at every parent node, an aggregate of its two children. Segment trees in Data Canopy are implemented as binary trees. The Data Canopy catalog implemented as a hash table stores pointers to all segment trees.

Segment trees are well-suited as a data structure for Data Canopy. This is because to

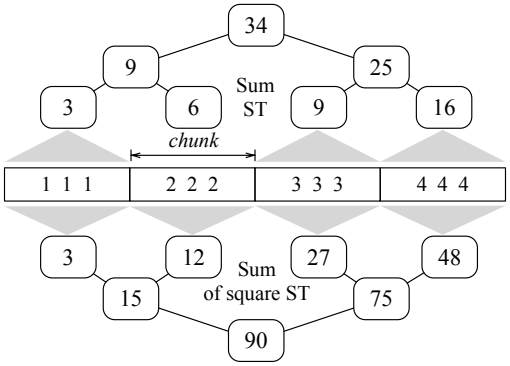


Figure 4.2: Example of the Data Canopy data structure with two segment trees (ST) and a chunk size of three.

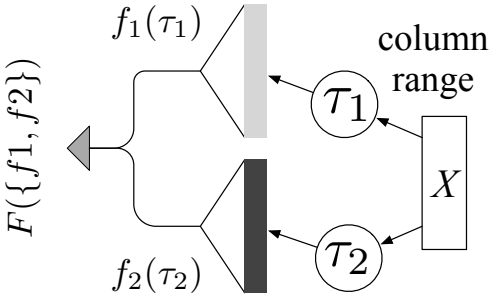


Figure 4.3: Data Canopy decomposes Statistics into basic aggregates to enable various forms of reuse.

synthesize queries that request for statistics over a data range, Data Canopy only needs aggregates over chunks that fall within that data range, and not their actual values. Consider Query 1 in our running example. Data Canopy stores basic aggregates over 12 values (daily basic aggregates). A query that requests weekly standard deviation only needs sum and sum of squares over 14 consecutive basic aggregates, and not their actual values. This way, Data Canopy can synthesize statistics in time complexity which is logarithmic in the number of chunks involved.

Data Structure Configuration. For every basic aggregate kept for every column, Data Canopy maintains a separate segment tree. Every leaf of this segment tree stores a basic aggregate value corresponding to a chunk. An example layout of the Data Canopy data structure over a single column is shown in Figure 4.2. In this example, Data Canopy holds two basic aggregates (sum and sum of squares), using two separate segment trees, one for each basic aggregate.

By having a separate set of segment trees for every column, we ensure that the internal nodes of each segment tree contain no surplus nodes (i.e., those that maintain aggregates across columns or across statistical measures). As a result, the overall memory requirement of Data Canopy as well as the size of the individual segment trees is mini-

Term	Description
c	Number of columns
r	Number of rows
h	Number of chunks
s	Chunk size (bytes)
v_d	Record size (bytes)
v_{st}	ST node size (bytes)
#	Cache line size (bytes)

Table 4.2: Data Canopy terms.

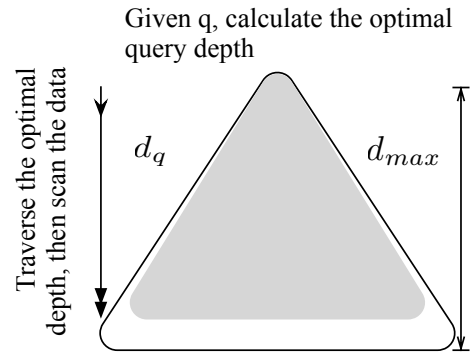


Figure 4.4: For each query, Data Canopy traverses the optimal depth d_q of the segment trees.

Options	Memory	Query/Update
ST per Data Canopy	$2 \cdot b \cdot c \cdot h - 1$	$O(\log b \cdot c \cdot h)$
ST per column	$2 \cdot b \cdot c \cdot h - c$	$O(\log b \cdot h)$
ST per statistic	$2 \cdot b \cdot c \cdot h - s$	$O(\log c \cdot h)$
ST per column per statistic (Data Canopy)	$2 \cdot b \cdot c \cdot h - b \cdot c$	$O(\log h)$

Table 4.3: Memory, access, and update cost of different configurations of segment trees (ST) storing b basic aggregates. The configuration used by Data Canopy (bottom) has the lowest query cost and memory usage.

mized. Also, since range queries are localized to a single column or a set of columns (for multivariate statistics) instead of the entire data set, we only have to search through a subset of the total segment trees, instead of one big segment tree corresponding to the entire data set. This arrangement still allows a data scientist or application to request for individual statistics and combine them in ways that make sense according to the domain and the data set. A comparison of the memory requirement and query cost of various possible configurations of segment trees is provided in Table 4.3. The configuration used in Data Canopy has the lowest query cost and memory usage.

Flexibility. The separation of segment trees allows for maximum flexibility in dynamic and exploratory workloads. There is no need to construct or even allocate memory for

the entire Data Canopy in advance. Instead, Data Canopy can easily be extended, by adding new segment trees, to cater for new columns or new basic aggregates.

Parallelism. The construction of Data Canopy can be aggressively parallelized as the process of calculating basic aggregates and storing them is an embarrassingly parallel one. To construct a univariate Data Canopy, the columns can be divided between the number of available hardware threads. Similarly, when constructing a multivariate Data Canopy, the segment trees for every column combination can be built independently.

4.4 Operation Modes

Depending on hardware properties, data size, and latency requirements, Data Canopy can operate in one of three modes: *offline, online,* and *speculative.*

Offline. In the offline mode, Data Canopy is built in advance. This mode is useful when users know the data and statistical measures of interest a priori and they can also wait until Data Canopy is built before they pose their first query. The offline mode builds the library of basic aggregates fully for a set of rows, columns, and statistical measures specified by the user.

Online. In the online mode Data Canopy populates the library of basic aggregates incrementally online during query processing. For every incoming query, Data Canopy generates and caches the basic aggregates needed for this query if they do not already exist in the library. As more queries are being processed, the library of basic aggregates becomes more and more complete and can reduce data access costs for future queries with higher probability.

The online mode can be combined with the offline mode. For example, a user may generate any portion of the Data Canopy for any part of the data offline (or generate as much as idle time allows) and then during query processing, Data Canopy operates in online mode to fill in the rest of the missing pieces.

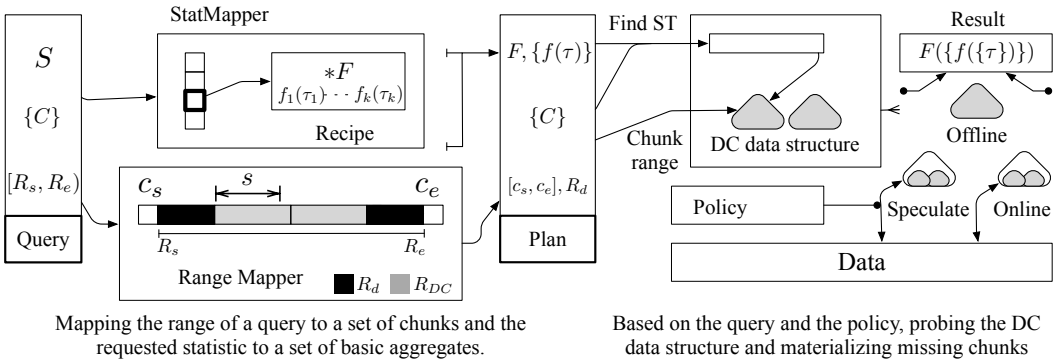


Figure 4.5: The lifecycle of a statistical query in Data Canopy.

Speculative. In the speculative mode, Data Canopy takes full advantage of moving the data through the memory hierarchy to generate more knowledge than what is strictly needed for the active query. Every time it scans any part of the data set to answer a query, it builds segment trees for all univariate statistics. We show that this imposes a modest CPU and memory overhead for the current query, and Data Canopy potentially avoids having to rescan the data for future queries for other statistics – trading a modest CPU and memory overhead now for I/O benefits later on. For example, when Data Canopy answers a mean query in speculative mode, it also builds a segment tree for sum of squares so that it is possible to later efficiently synthesize the variance and standard deviation.

4.5 Query Processing

We now explain how Data Canopy uses its library of basic aggregates to synthesize the results of statistical queries. We use terms from Table 4.2.

Query. In Data Canopy, a query is defined by the set $Q = \{\{C\}, [R_s, R_e), S\}$, where $\{C\}$ is the set of columns targeted by the query; R_s and R_e define the query range i.e., the two positions on the column set C on which a statistic is requested; and S is the statistical measure to be computed. From our running example, Query 2 (mean

temperature for the third week) can be represented as $Q_t = \{C_t, [336, 504), \mathit{mean}\}$. Figure 4.5 depicts the steps taken to process a query. The first step is to convert the query into a plan. To achieve this, the query range is mapped to a range of chunks, and the statistical measure is mapped to a set of basic aggregates.

Mapping Query Range to Chunks. Data Canopy first maps the query range $[R_s, R_e)$ to a set of chunks $[c_s, c_e]$, such that the whole query range is covered. This process is depicted on the left side of Figure 4.5, where the query range (shown in black and grey) is mapped to the corresponding chunks. Given the mapping, we can now distinguish between two parts of the query range. The first part of the query range R_{DC} (shown in grey) aligns perfectly with the boundaries of the existing chunks. In this case, Data Canopy can fully use the basic aggregates of these chunks to synthesize the result. The second part of the query range R_d (shown in black) at the two end-points of the query range might or might not align with the existing chunks. Data Canopy has to scan the two chunks at the end-points of the query range to compute basic aggregates for R_d. We call this part of the query range that always requires access to base data the residual range. When Data Canopy operates in online mode, it may be that it has to access more than two chunks so as to populate any missing chunks in any part of the query range, not just at the end points.

Mapping Statistic to Basic Aggregates. The next step is to map the requested statistical measure S to the corresponding set of basic aggregates $\{f(\tau)\}$ and a function F to combine these basic aggregates. This is achieved by the *StatMapper* as shown in Figure 4.5. For every statistical measure supported by Data Canopy, the StatMapper stores a complete *recipe* to synthesize that statistic from basic aggregates.

The StatMapper is implemented as a hash table, where the keys are identifiers of statistical measures and each key corresponds to a recipe. The recipe is a data structure that contains a list of basic aggregates $\{f(\{\tau\})\}$ required to synthesize the statistical measure S as well as a pointer to a function that operates on and combines the basic aggregates as defined by F. Overall, Data Canopy converts a query Q into a plan P, making the following set of mappings:

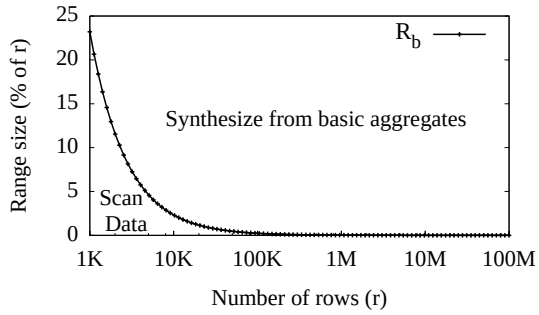


Figure 4.6: As the number of rows in the data set increases, a greater proportion of the total queries is answered through basic aggregates.

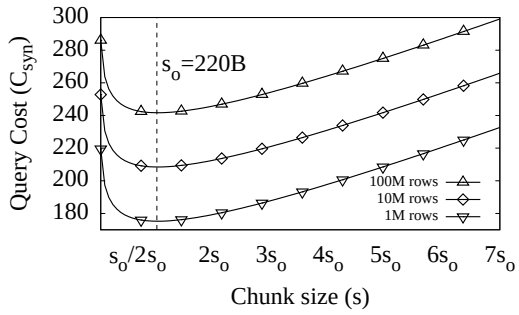


Figure 4.7: Query cost, a convex function of the chunk size, is minimized at the optimal chunk size s_o. Here #=64B, b=5, and k=2, s_o = 220B.

$$\{\{C\}, [R_s, R_e), S\} \rightarrow \{\{C\}, [c_s, c_e], R_d, \{f(\{\tau\})\}, F\}$$

Evaluating the Plan. The plan is passed on to the evaluation engine, where the result is synthesized based on the current policy and state of Data Canopy (right side of Figure 4.5).

If Data Canopy is operating in the offline mode, all basic aggregates have been precomputed and there is no need to touch the base data except to evaluate the residual range R_d. In this mode no new basic aggregates are added as a result of query processing. In the online and the speculative mode, some of the required basic aggregates (for some chunks) might not be computed and stored already. In such cases, Data Canopy accesses base data to evaluate basic aggregates on those chunks, and they are stored in Data Canopy. Finally, when all basic aggregates required for the current query are fetched and/or materialized, they are passed to function F to generate the result.

4.5.1 Analyzing Query Cost

We formalize the cost of answering a query when both Data Canopy and data fit in memory (we model the out-of-memory cost in §4.8). This cost is modeled in terms of

the amount of data accessed (cache lines).

We consider a query q for a statistic S over a data range. The statistic S is defined over k different columns, and it is composed of b total basic aggregates i.e., it accesses b segment trees. For instance, in the case of a variance query, $b = 2$ (sum and sum of squares) and $k = 1$ (univariate statistic), whereas for a correlation query $b = 5$ (sum and sum of squares of both columns and sum of products) and $k = 2$ (bivariate statistic).

Let C_{syn} be the cost of answering query q. This cost is divided in two parts: (1) probing b segment trees, and (2) scanning the residual ranges of k columns. We denote these costs as C_{st} and C_r respectively. The total cost is:

$$C_{syn} = C_{st} + C_r$$

First, we model C_{st}. To answer a query q, Data Canopy traverses b segment trees. The number of leaves in each segment tree is $\frac{r \cdot v_d}{s}$, where r is the number of rows, v_d is the record size (in bytes), and s is the chunk size (in bytes). Moreover, the cost of probing a segment tree with n leaves is at most $2 \log n$ cache line reads (Zheng et al.) (as a node fits in a cache line). Hence, we can express C_{st} as follows:

$$C_{st} = 2 \cdot b \cdot \log_2 \left(\frac{r \cdot v_d}{s} \right) \tag{4.2}$$

We now model C_r. A query on k columns has to scan at most $2k$ chunks i.e., at the end points of the query range. The cost of scanning a chunk is $\frac{s}{\#}$. We get the following formula for C_r:

$$C_r = \frac{2 \cdot k \cdot s}{\#} \tag{4.3}$$

Using Equation 4.2 and 4.3, the total query cost becomes:

$$C_{syn} = \frac{2 \cdot k \cdot s}{\#} + \left(2 \cdot b \cdot \log_2 \frac{r \cdot v_d}{s} \right) \tag{4.4}$$

For simplicity of presentation, here we do not distinguish between the cost of a cache miss (traversing the linked segment trees) and a cache hit (scanning a sequential residual range). We study the effects of these hardware dependent parameters when we tune and verify the chunk size in Section 4.10.7.

Synthesize or Scan. For queries with a small range, Data Canopy directly scans the data if this results in a smaller query cost compared to traversing the segment trees and synthesizing the answer. We describe below how this optimization decision is made.

The cost of scanning the full query range of size R, C_{scan} can be expressed as:

$$C_{scan} = \frac{R \cdot v_d}{\#} \tag{4.5}$$

Now we calculate the boundary query range size R_b, where C_{scan} becomes equal to C_{syn}. Below R_b, answering the query by scanning the complete query range is faster than synthesizing it from basic aggregates. Using Equation 4.4 and 4.5, we get:

$$R_b = \frac{2 \cdot k \cdot s}{v_d} + \frac{2}{v_d} \cdot \# \cdot b \cdot \log_2\left(\frac{r \cdot v_d}{s}\right) \tag{4.6}$$

Data Canopy answers a query with range size R from basic aggregates when $R > R_b$, otherwise it answers the query by scanning the full query range. Figure 4.6 shows how R_b (as a percentage of the number of rows r) decreases as r increases. This shows that as the number of rows in the data set increases a greater proportion of total queries is answered through basic aggregates. Here $\# = 64B$, $b = 5$, $k = 2$, and $v_d = 4B$.

4.6 Selecting the Chunk Size

We now explain how Data Canopy selects the chunk size to optimize query performance.

Optimal Chunk Size. The chunk size has opposite effects on the cost of scanning the residual range C_r and the cost of traversing segment trees C_{st}. Increasing the chunk size, results in an increase of C_r as the residual range increases. On the other hand,

increasing the chunk size decreases C_{st} as the size of segment trees shrinks. As a result, C_{syn} is a convex function of the chunk size and has a global minimum i.e., there is an optimal chunk size s_o that optimizes overall query performance. The convex behavior of the query cost is shown in Figure 4.7 ($\# = 64B, b = 5, k = 2$). To obtain a closed-form expression for the optimal chunk size s_o, we differentiate C_{syn} with respect to s:

$$s_o = \frac{b \cdot \#}{k \cdot \ln 2} \tag{4.7}$$

The optimal chunk size s_o depends only on properties of the hardware (i.e., cache line size) and the type of requested statistic (i.e., the ratio between the number of segment trees and the columns that are scanned for the residual range). This is because the optimal chunk size strikes a balance between the number of cache lines accessed when scanning the base data (for the residual range) and when traversing the segment trees.

Optimal Chunk Size and R_b. Observe from Equation 4.6 that $s < R_b, \forall r \geq s$. In other words, any chunk size (including the optimal chunk size s_o) is always smaller than the boundary range size R_b below which a given query is answered by scanning the range. A corollary of this observation is that independent of the workload the chunk size should not be below s_o. This is because Data Canopy will answer any query with a smaller range size than s_o by directly scanning the range instead of traversing the segment trees (because this is faster i.e., it incurs fewer cache line reads).

Selecting the Chunk Size. By default, Data Canopy sets the chunk size s_{DC} to the lowest value of the ratio $\frac{b}{k}$. This value is 1 (for b=k=1) and allows Data Canopy to store just enough information (enough depth in the segment trees) to be optimal for queries that access the least amount of segment trees (e.g., mean, max, min etc.). Hence, to set the default chunk size, Data Canopy needs no prior knowledge of the workload.

Workload Adaptivity. To ensure optimal performance for queries with $\frac{b}{k} > 1$ (i.e., those that access more than one segment trees), Data Canopy makes an adaptive decision and traverses shorter paths in the segment trees. This strategy is shown visually in Figure 4.4. Given a query q, Data Canopy analytically computes the optimal chunk size for this query s_q using Equation 4.7. Then it calculates the optimal depth of the

segment tree for q:

$$d_q = \log_2 \left(\frac{r \cdot v_d}{s_q} \right)$$

Data Canopy goes only as deep as d_q in the segment trees, and then scans the residual range (now up to a size of $2 \cdot k \cdot s_q$). This strategy ensures that each query achieves optimal performance by minimizing the data (cache lines) it has to read.

Overall, Data Canopy builds segment trees with a chunk size that guarantees optimality for queries that need to access a single segment tree only (i.e., d_{max}) and can afford to do more cache misses going all the way to the leaves of the segment tree. For queries that will access more segment trees, though, (and thus they will incur more cache misses) Data Canopy adaptively gets out of the segment tree traversal sooner (i.e., at d_q) reverting on sequentially scanning more data chunks and thus achieving an optimal balance tailored to each individual query. This optimization comes from the fact that segment trees are binary trees and every node we read when traversing the tree leads to a cache miss. As such there is a point when reading a cache line full of useful data (when scanning data chunks) becomes better than traversing a binary tree. Other directions, one may explore here, as alternatives to the optimization we propose, is the study of a more cache conscious layout of the segment trees where every cache miss would bring a cache line full of useful tree data.

4.7 Memory Footprint

Data Canopy's memory requirement depends on: (i) the types of statistical measure it maintains, (ii) the chunk size, and (iii) the data size. For a given set of statistics S, we define the Data Canopy footprint $\mathscr{F}(S)$ as the number of segment trees per column required to synthesize S on the entire data set. The size (in bytes) of a full segment tree with the optimal chunk size s_o and node size v_{st} is given by $v_{st} \cdot (2 \cdot \frac{r \cdot v_d}{s_o} - 1)$. Hence, the total size of a complete Data Canopy (in bytes) on c columns is:

$$|DC(S)| = c \cdot v_{st} \cdot (2 \cdot \frac{r \cdot v_d}{s} - 1) \cdot \mathscr{F}(S) \tag{4.8}$$

We define the Data Canopy footprint with respect to both a single statistic and a set of statistics, as the number of basic aggregates per column required to synthesize all instances of that statistic or set of statistics from Data Canopy. Below we elaborate how it applies to univariate and bivariate statistics.

Univariate Statistics. The Data Canopy footprint of univariate statistics is independent of the number of columns c. This is because to compute univariate statistics on a column, we require no information from other columns. For example, the Data Canopy footprint of mean is 1 because we need to keep only the sum for every column to synthesize the mean. Similarly the Data Canopy footprint of variance and standard deviation is 2.

Bivariate Statistics. The Data Canopy footprint of bivariate statistics depends on the number of columns c as they require information from pairs of columns. For example, to synthesize all pairwise correlations, we need sums and sums of squares of all c columns as well as $\frac{c \cdot (c-1)}{2}$ sums of pairwise products i.e., a total of $\frac{2+(c-1)}{2}$ basic aggregates are stored for every column.

Set of Statistics. The Data Canopy footprint is similarly defined for a set of statistics. For example, the Data Canopy footprint of mean and variance is 2 whereas the Canopy footprint of standard deviation, mean, and correlation is $\frac{2+(c-1)}{2}$.

Using terms from Table 4.2, we define the size of Data Canopy storing a set of statistics S as follows:

$$|DC(S)| = c \cdot (2 \cdot h - 1) \cdot \mathscr{F}(S) \cdot v_{st}$$

Composabilty. Here we define the concept of composability, which can be used to characterize the reusability of the basic aggregates cached by Data Canopy. Composability is the extent to which basic aggregates are shared by the set of statistics S supported by Data Canopy. Formally, it is the ratio between the number of basic aggregates shared by all members of S and the total number of basic aggregates required to synthesize S.

Let $\mathscr{B}(S)$ be the set of basic aggregates required to synthesize a statistic S, then the composability of S, given by $\mathscr{C}(S)$ is:

$$\mathscr{C}(S) = \frac{\bigcap_{i=1}^{i=|S|} \mathscr{B}(S_i)}{\bigcup_{i=1}^{i=|S|} \mathscr{B}(S_i)}$$

For instance, the composability of $S = \{\text{mean, variance, standard deviation}\}$ is one-half. $\mathscr{C}(S)$ is zero when none of the statistics share any of the basic aggregates. On the other hand, $\mathscr{C}(S)$ is one when the same set of basic aggregates can be used to compute every member of S. A highly composable set of statistics will result in better reusability and lower memory requirement.

4.8 Out-of-Memory Processing

Now we introduce a three-phase eviction policy that maintains good performance guarantees as the data size and the size of Data Canopy exceeds main memory capacity. The high level idea is that Data Canopy maintains a cache of data pages, which are evicted when there is memory pressure and reloaded if needed. Similarly, parts of Data Canopy are also evicted and reloaded if needed. This policy captures both the case when data does not fit in memory and the case when Data Canopy does not fit in memory.

Phase 1. During the first phase, as main memory runs out, Data Canopy shrinks horizontally by removing one layer of leaf nodes from every segment tree in a round-robin fashion. This is equivalent to doubling the chunk size. Both data and Data Canopy still fit in main memory, and so the system maintains good performance (i.e., query processing is in the order of hundreds of microseconds). If there is more memory pressure and the chunk size exceeds the size of a page (4KB to 64KB), Data Canopy stops shrinking and moves on to Phase 2.

Phase 2. During Phase 2, Data Canopy maintains data pages in memory only as a cache of frequently accessed data. It evicts data pages from main memory using an LRU policy. Query cost remains low since each query has to touch at most $2k$ pages to scan the residual range, where k is the number of columns referenced by a query.

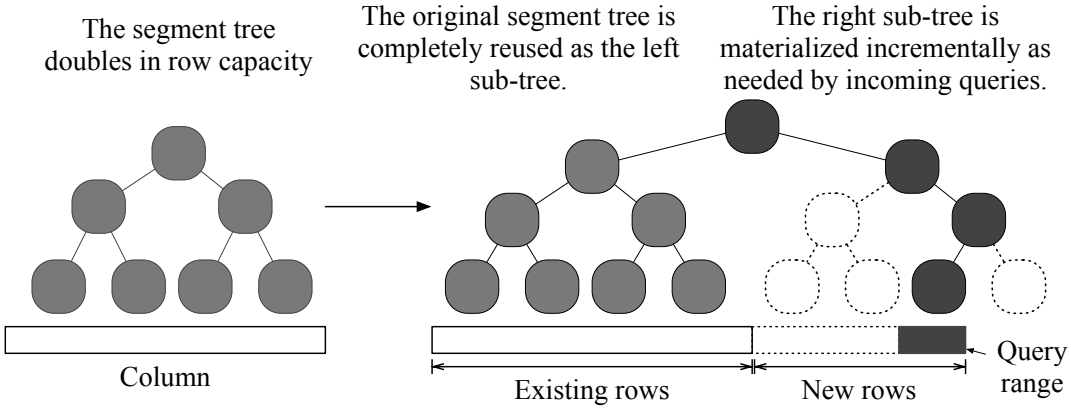


Figure 4.8: Data Canopy adaptively handles new data (rows).

For example, a correlation query needs to access at most two columns and thus touches at most four pages, which takes approximately 40 ms on modern disks. Moreover, for frequently accessed chunks, the cache prevents a query from going to disk.

Phase 3. In the extreme case, when none of the data can fit in memory, we reach the scenario, where parts of Data Canopy also need to be evicted. In this case, Data Canopy evicts whole segment trees using an LRU policy. These segment trees are spilled to disk and reloaded if needed. To make it easy when reloading segment trees from disk that may refer to potentially dirty chunks (updated), we keep an in-memory bit vector for each segment tree, which marks dirty chunks (1 bit per chunk). If memory pressure continues, bit vectors are also dropped along with the on-disk segment trees.

Offline Mode and Memory Pressure. When Data Canopy is set to offline mode it is given a set of data (row and columns) and a set of statistics to be precomputed. Data Canopy first computes the overall memory footprint that the resulting structure will have and if it exceeds available memory, Data Canopy has to operate immediately in Phase 3. Before doing so, Data Canopy first gives the user a warning and option if they want to reduce the amount of data or statistics to be included so that it fits in the memory budget. Otherwise, Data Canopy proceeds in Phase 3.

4.9 Handling Updates

We now discuss how Data Canopy handles insertions, updates, and deletes. Data Canopy handles updates incrementally to avoid overheads during online exploration.

Inserting Rows. When new rows are inserted and the new total number of rows exceeds the existing capacity of Data Canopy, then Data Canopy needs to expand. It does so by doubling the capacity of its segment trees without doubling the size immediately. This means that a root is added in each segment tree with the previous root as a left child and a new empty right child (and subtree). This results in effectively no immediate memory overhead. Data Canopy then populates the new right sub-tree adaptively only when and if the new rows are queried (Figure 4.8).

Inserting Columns. When a new columns is added, Data Canopy needs to simply add this column in its catalog. Given that columns are treated independently there is no further complexity resulting from the addition of a new column. As data in the new column is queried, Data Canopy allocates segment trees for this column and then populates them incrementally.

Updating Rows. When a record x at row r of column c is updated, Data Canopy first retrieves the old value x_{old} of x and uses it along with the new value x_{new} of x to update all segment trees that involve column c. For each segment tree, Data Canopy looks up the basic aggregate y_{old} for the chunk where row r resides, and it updates it as follows[1]: $y_{new} = y_{old} - \tau(x_{old}) + \tau(x_{new})$.

Assuming a univariate segment trees on column c, the cost of updating them is $a \cdot \log_2 \frac{r \cdot v_d}{s}$ (where $\log_2 \frac{r \cdot v_d}{s}$ is the depth of the segment trees). Moreover, assuming b bivariate segment trees on column c, the cost of updating them is $b \cdot \log_2 \frac{r \cdot v_d}{s} + b$. The additive b term derives from the fact we need to fetch one value from another column per segment

[1]More generally, we update y using the aggregation function F and its inverse F^{-1} as follows: $y_{new} = f\left(f^{-1}\left(\tau(x_{old}), y_{old}\right), \tau(x_{new})\right)$.

tree to adjust the sum of products. The overall update cost C_{update} is:

$$C_{update} = 2 \cdot (a + b) \cdot \log_2 \frac{r \cdot v_d}{s} + b \tag{4.9}$$

Deleting Rows. Data Canopy deletes rows in-place using a standard technique for fixed-size slotted pages, where the granularity of a page is the chunk. Each chunk has a counter that keeps track of the number of valid rows in a chunk, and the valid rows are placed first in the chunk. When a row is deleted, we replace each deleted value x_{old} with the last valid value in the chunk, and we decrement the counter.

To update the segment trees, we probe all of them for the basic aggregate for the chunk of the deleted row and update it as follows[2]: $y_{new} = y_{old} - \tau(x_{old})$. In addition, we maintain one *invalidity segment tree* per table that keeps track of the number of invalid entries per chunk for subsequent statistical queries, as we can no longer assume that each chunk is full. The cost model is the same as for updates with one more additive term of $2 \cdot \log_2 \frac{r \cdot v_d}{s}$ for updating the invalidity segment tree: $C_{delete} = C_{update} + 2 \cdot \log_2 \frac{r \cdot v_d}{s}$.

4.10 Experimental Analysis

We now demonstrate that Data Canopy accelerates statistical analysis and machine learning algorithms.

Experimental Setup. All experiments are conducted on a server with an Intel Xeon CPU E7-4820 processor, running at 2 GHz with 16 MB L3 cache and 1 TB of main memory. This server machine runs Debian "Jessie" with kernel 3.16.7 and is configured with a hard disk of 300GB operating at 15KRPM. We implemented Data Canopy from scratch in C++ compiled with gcc version 4.9.2 at optimization level 3. The current prototype supports three univariate statistics: mean, variance, and standard deviation; and two bivariate statistics: correlation and covariance.

We compare the performance of Data Canopy with two widely used statistical packages: NumPy (num 2013) in Python and Modeltools (Foundation 2016) in R. Also, we show

[2]More generally, we apply: $y_{new} = f^{-1}(\tau(x_{old}), y_{old})$.

Workload	Column Dist.	Range Size	Repetition
U	Uniform	*Unif(5,10)* %	low
Z	Zipfian	*Unif(5,10)* %	moderate
U_+	Uniform	Zoom-in	high
Z_+	Zipfian	Zoom-in	very high

Table 4.4: Evaluation workloads.

how Data Canopy compares to MonetDB (Boncz et al. 1999). In addition to these systems, we compare Data Canopy against our own statistical system *StatSys*. StatSys shares the code base with Data Canopy, but it has none of the design concepts that allow Data Canopy to synthesize statistics from basic aggregates; instead, it needs to fully compute each query from scratch.

Benchmark. There are no standard benchmarks for exploratory statistical analysis. To test Data Canopy we develop a benchmark that captures a wide range of core scenarios and stress tests Data Canopy's capability to reuse data access and computation.

We generate exploratory statistical analysis pipelines as sequenc- es of queries. Each query requests to compute a statistical measure on a range over a data column (or a set of data columns for multivariate statistics). The benchmark consists of four distinct workloads generated by varying two parameters: the probability with which queries are distributed over columns and the distribution of query range sizes. These workloads are summarized in Table 4.4. We investigate two different distributions of queries over columns: column-uniform (U and U_+) and column-zipfian (Z and Z_+). In the column-uniform workloads, queries are equally divided between all columns. In the column-zipfian workloads, queries are divided over columns conforming to the zipfian distribution (s=1) i.e., the column with the highest number of queries has twice as much queries in the workload as compared to the column with the second highest.

Similarly, we investigate two different distributions for the query range sizes. In the range-uniform workloads (U, Z), the range sizes are uniformly distributed between 5 and 10 % of the total column size. The range-zoom-in workloads (U_+, Z_+) emulate a case where data scientists progressively zoom into the data set increasing the resolution at which statistics are computed. In this case, the range size follows a sequence, where the first query is over an entire range. All subsequent pairs of queries divide the range

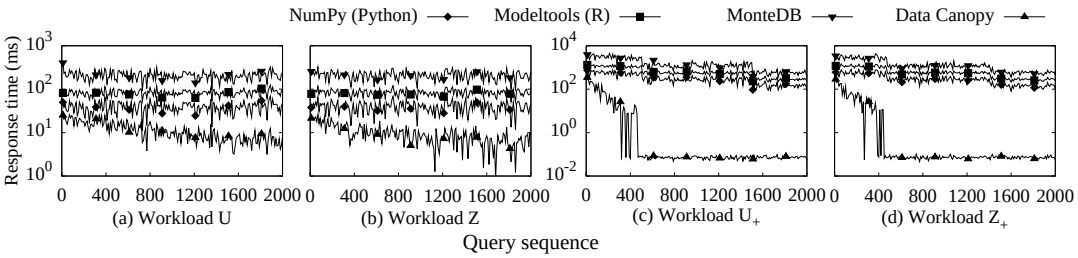


Figure 4.9: Data Canopy, in online mode, out performs state-of-the-art systems across a variety of workloads for exploratory statistical analysis by being able to incrementally improve its performance and minimize data access.

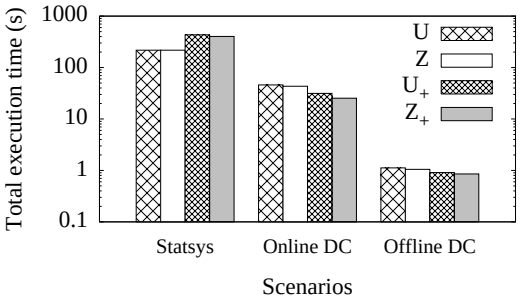


Figure 4.10: Online and offline Data Canopy result in one and two orders of magnitude improvement respectively.

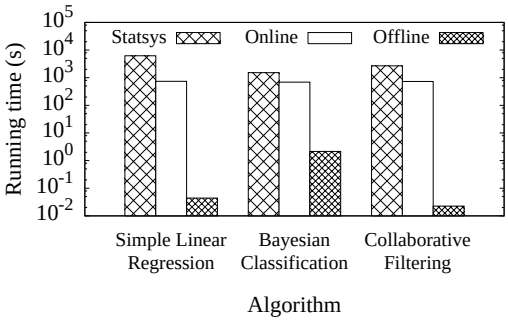


Figure 4.11: Data Canopy accelerates core machine learning classification and filtering algorithms.

of previous queries into two equal parts, then compute statistics on both. Then we randomly pick one of these parts to continue doing the same. For example, zoom-in over a range of size 100 can be the sequence: $\{[0,100), [0,50), [50,100), [50,75), \ldots\}$.

These workloads allow us to test Data Canopy with different kinds of repetition. They map to patterns followed by data scientists during data exploration: The initial phase of exploratory analysis, often classified as the foraging phase (Battle et al. 2016; Pirolli and Card 2005), exhibits patterns similar to column-uniform workloads. This is when data scientists compute statistics uniformly over multiple columns. Over time, the analysis focuses on a smaller set of columns (column-zipfian workloads), and requests for more detailed information (range-zoom-in workloads) (Battle et al. 2016).

4.10.1 Reuse in Exploratory Statistical Analysis

In our first experiment we compare Data Canopy against state-of-the-art systems and we demonstrate its ability to reuse data access and computation. We set-up this experiment as follows: The data set contains 40 million rows and 100 columns. Each column is populated with double values randomly distributed in $[-10^9, 10^9)$. The total data size is 32GB. Data Canopy is automatically configured with the optimal in-memory chunk size. For our experimental system, this results in a chunk size of 256 bytes or 32 data values (in Section 4.10.7 we verify the chunk selection model). Data Canopy operates in the online mode, which provides an apples-to-apples comparison across all systems as it assumes no preprocessing steps.

Figure 4.9 shows the results for all four workloads. Each one of the four graphs in Figure 4.9 corresponds to one of the workloads in Table 4. Each graph depicts the evolution of the query performance (response time on the y-axis) as the query workload evolves, i.e., as we run more exploratory queries (x-axis). In total we run 2000 queries for each workload. Each graph shows the performance of NumPy, R, MonetDB, and Data Canopy.

The main observation across all graphs in Figure 4.9 is that while all state-of-the-art systems maintain a relatively constant behavior across all workloads, Data Canopy improves as it processes more queries. The y-axis is logarithmic and depicts the response time per query. For example in Figure 4.9(a) after just a hundred completely uniform queries, the average response time of Data Canopy is 1.9× lower than NumPy and 11.4× lower than MonetDB. After 2000 queries, the performance improvement per query goes up to 6.7× and 34.5× respectively. Thus, in most cases Data Canopy results in an overall benefit (during an exploration path, i.e., over a sequence of queries) of multiple orders of magnitude. The longer the exploration path the bigger the benefit.

In addition, Data Canopy is faster than all other systems even for the very first query across all workloads in Figure 4.9. This is because contrary to NumPy and R, Data Canopy is a tailored C++ implementation for statistics. MonetDB is a performant analytical system but it is not tailored for statistics.

Similar observations hold for Figures 4.9(c) and 4.9(d) where the workloads exhibit zoom-in patterns. In these workloads, the range size decreases by half after the first 500 queries. Then, it decreases by half every 1000 queries. This constant decrease in range sizes is reflected in the response times of all systems. In other words, all systems can improve nearly linearly to the size of the range on which statistics are computed. This is because they simply do computations on fewer data items. On the other hand, Data Canopy improves drastically by being able to reuse previous data accesses and computations. For all queries after the first 500 queries, the average response time goes down to sub-milliseconds. Even during the first 500 queries, there is a continuous sharp improvement in Data Canopy's response time. In both workloads, Data Canopy is completely built at the end of the first 500 queries, and all future queries are directly synthesized from the basic aggregates within Data Canopy.

For all systems and for all these experiments we make sure that all data is hot in memory before we query it. This is the least favorable scenario for Data Canopy as its goal is to reduce data access costs.

Data Canopy Scenarios. Next we evaluate the offline and online modes of Data Canopy. In addition, we compare against StatSys, which effectively uses the Data Canopy code to compute statistics but does not cache and reuse basic aggregates.

The set-up of this experiment is exactly the same as before. The results are shown in Figure 4.10. This time we report the cumulative response time to run all queries. For all workloads Data Canopy results in significant benefits over the no reuse approach of StatSys (up to one order of magnitude i.e., 4.7× to 15.8×). If we can allow to precompute the library of basic aggregates up front this brings yet another benefit of two orders of magnitude (194× to 470.8×). In this scenario all queries are directly synthesized from Data Canopy (each query may at most scan two chunks at the boundaries of its range). Overall, the improvement is bigger for range-zoom-in workloads (U_+ and Z_+). This is because for these workloads the first query on every column results in a complete scan, due to which basic aggregates required for future queries on that column are already computed. Overall, Data Canopy is effective in both online and offline mode bringing drastic improvements in response time.

4.10.2 Accelerating Machine Learning

We now show how Data Canopy accelerates core machine learning classification and filtering algorithms. Specifically we study linear regression, bayesian classification, and collaborative filtering (Bishop 2006). All three algorithm can utilize statistics (basic aggregates) cached in Data Canopy as primitives. The set-up is the same as in previous experiments (40 million rows and 100 columns) and we run each of the algorithms on the entire data set as follows: (i) Simple linear regression is ran on all pairs of columns, (ii) A gaussian naive bayes classifier is trained on the entire data set. In this case, the rows in the data set are divided between 40 different classes (one million samples per class), (iii) Collaborative filtering (using correlation as the similarity measure) is ran on the entire data set.

Figure 4.11 shows the performance of these three machine learning algorithms with Statsys (brute force), online, and offline Data Canopy. We observe that online Data Canopy (no preprocessing step) results in up to 8× improvement. This is because running these algorithms results in repetitive calculation of statistics. Furthermore, if there is enough idle time to build Data Canopy offline, we observe up to six orders of magnitude improvement in running time for simple linear regression and collaborative filtering and three orders of magnitude improvement for bayesian classification. The lower improvement for bayesian classification is due to the fact that we have to compute statistics for every class in the data set (i.e., 40 times more queries and each query results in scan of up to two chunks per column at the end-points of the query range).

4.10.3 Scalability

Here we show that Data Canopy scales with the data size (number of columns and rows), hardware contexts, and queries.

Scaling with Number of Rows. First, we show how Data Canopy scales when we increase the number of rows in the data set. The set-up is the same as in previous experiments. This time we vary the rows from 100 million to one billion.

Figure 4.12 reports the results. It depicts the cumulative time to run all four workloads.

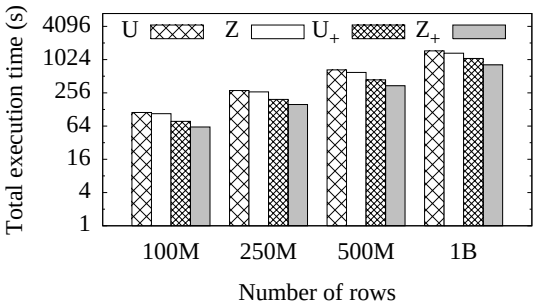


Figure 4.12: Data Canopy scales almost linearly with the number of rows in the data set for all workloads.

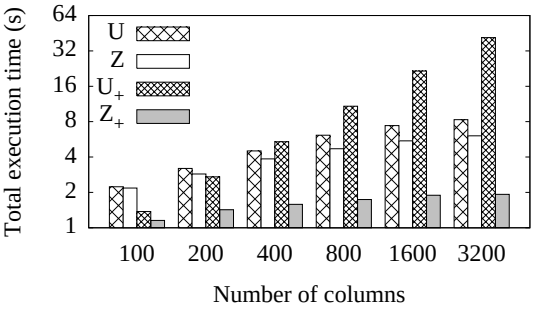


Figure 4.13: Data Canopy scales with the number of columns resulting in sub-linear increase in query execution time.

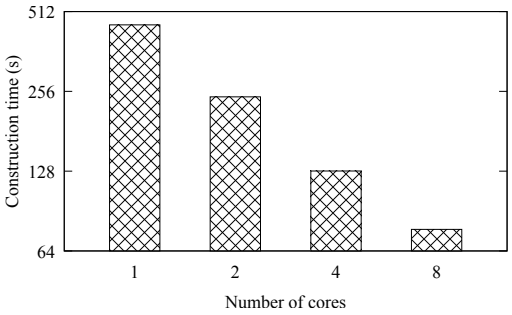


Figure 4.14: The construction of Data Canopy scales linearly with the cores.

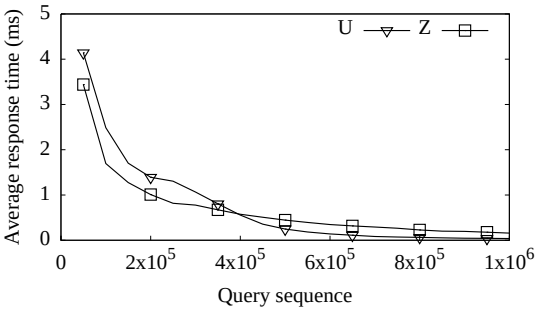


Figure 4.15: As we increase the number of queries, the query response time continuously goes down (up to 190×).

As we increase the number of rows from 100 million to 250 million, the total execution time increases by 2.51x (average across all workloads) i.e., an approximately linear increase in execution time. As we double the number of rows beyond 250 million, the trend diverges slightly from a linear trend. The increase in cumulative response time as we increase the number of rows from 250 million to 500 million and from 500M to 1 billion is 2.26x and 2.3x respectively. This super-linear increase in cumulative response time is due to the fact that with more rows, the size of the query range (unif(5,10)% of r) increases. This results in more chunks being added to the Data Canopy data structure, for every query that is executed. The overhead of adding these chunks results in this super-linear increase in the overall response time.

Scaling with Number of Columns. Now, we show how Data Canopy scales as we vary the number of columns from 100 to 3200. In this experiment, the number of rows is fixed to one million.

Figure 4.13 reports the cumulative time to run all four workloads. As we double the number of columns, we see an average increase of 1.68x and 1.22x in the total execution time for the uniform (U and U_+) and zipfian (Z and Z_+) workloads respectively. In all cases the execution time increases in a sub-linear fashion. For uniform workloads there is a higher increase in the total execution time because they target all columns equally and it takes longer to populate the library of basic aggregates. For the zipfian workloads, since the columns are targeted following a zipfian distribution, increasing the number of columns does not substantially affect the overall execution time – columns that are frequently accessed will have their corresponding library of basic aggregates completely materialized.

Overall, Data Canopy scales in a robust way, being able to absorb the increased amount of rows and columns.

Scaling with HW contexts. We first show the construction of Data Canopy scales as we increase the number of cores. We construct a complete Data Canopy (on 40 million rows and 100 columns) as we increase the amount of cores. Figure 4.14 shows that the construction time of Data Canopy goes down linearly with the number of cores. This is because the basic aggregates can be computed and cached completely in parallel.

Scaling with the Number of Queries. We now show how Data Canopy scales when we increase the number of queries. We keep the same overall setting as before. We report scaling results only for the range-uniform workloads (U and Z). This is because for the range-zoom-in workloads (U_+ and Z_+), Data Canopy is completely built after the first 500 queries. Thus, all future queries are synthesized directly from Data Canopy (with minor data accesses to compute residual ranges), and the average response time remains constant thereafter.

Figure 4.15 shows the results. To make it easier to interpret, we report the average response time for every sequence of $50K$ queries. The more queries are processed, the more Data Canopy improves. For example, the last query takes up to 190.9× less time

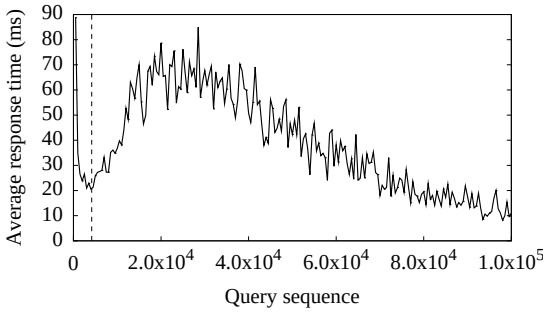


Figure 4.16: Data Canopy gracefully handles memory pressure, keeping query processing time within an interactive range.

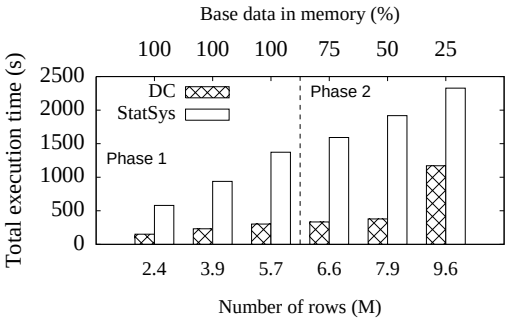


Figure 4.17: In memory-constrained settings, Data Canopy provides 4× performance improvement over Statsys.

to compute than the first one. Toward the second half of the query sequence, the pace of improvement decreases as more queries can be synthesized directly from the library of basic aggregates without accessing the base data. The initial improvement in average response time is higher for workload Z as compared to workload U because queries exhibit more locality in the first one; once the library of basic aggregates is constructed, though, performance is nearly the same for both workloads as all queries are resolved directly from this library with only minor access to base data (for residual chunks).

4.10.4 Handling Memory Pressure

We now demonstrate that Data Canopy gracefully handles memory pressure resulting from: (i) processing queries (that increases the size of materialized basic aggregates) and (ii) increasing the size of base data.

Memory Pressure From Processing Queries. For this experiment we allow a memory budget of 8GB. The size of the data is set to 7.2 GB (90 columns, 10 million rows, 8 bytes record size). This means that initially the entire data set fits in main memory. Data Canopy operates in online mode which means that initially it has zero memory footprint and it grows as more queries arrive. We run a sequence of queries from the U workload. This implies that Data Canopy incrementally materializes new segment trees, increasing memory pressure.

Figure 4.16 shows how the average response time of Data Canopy evolves as memory pressure increases. The dotted line depicts the point beyond which Data Canopy operates in Phase 2 of the out-of-memory policy i.e., some data is now accessed from disk. We observe that as Data Canopy enters Phase 2, there is an initial increase in query response time. This is because Data Canopy is still being built, and every query may result in a scan of data on disk. However, as the query sequence evolves and Data Canopy materializes further, the query response time decreases. Now, Data Canopy scans at most two chunks per query.

Memory Pressure From Increased Data Size. Now we analyze the scenario, where the memory pressure is due to an increase in the size of base data. Specifically, we show how Data Canopy compares with Statsys (our baseline system that shares the codebase with Data Canopy but always compute statistics from data instead of basic aggregates). We set up an experiment with 8GB of main memory and Data Canopy operates in the online mode. The number of columns is fixed to 100 and we vary the number of rows to test the performance of Data Canopy across different stages of Phase 1 and 2 of out-of-memory policy.

Figure 4.17 shows the total execution time of 10K queries from the U workload under different memory pressures. In Phase 1, Data Canopy remains consistently 4× faster than Statsys. As the memory pressure builds up and Data Canopy transition to Phase 2, it continues to give a performance improvement of 4× even when only 50 percent of the data fits in memory. Under extreme memory pressure (only 25 percent of the data fits in memory) Data Canopy still results in 2× performance improvement.

4.10.5 Memory Footprint and Feasibility

We discuss the memory footprint of Data Canopy in two scenarios: (1) when it is built with the optimal in-memory chunk size (256 bytes for our experimentation system) and (2) when, under memory pressure, it operates in Phase 2 of the out-of-memory policy (the chunk size grows to 64KB). These two scenarios correspond to the maximum and the minimum memory footprint of Data Canopy respectively. The experiment is on 100 columns and 40 million rows. Each node in Data Canopy is 8B. The analysis is

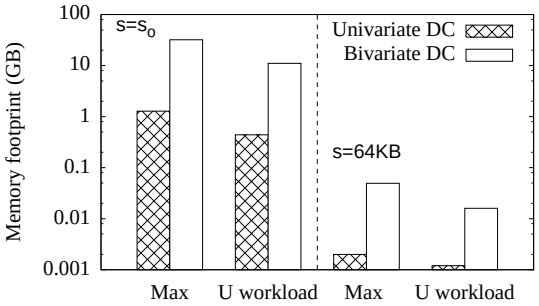


Figure 4.18: Under memory pressure, Data Canopy can vary its chunk size between the memory-optimized and disk-optimized size.

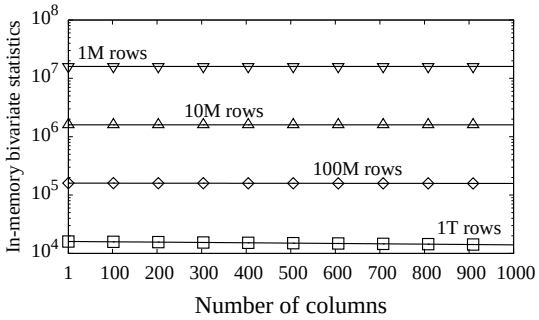


Figure 4.19: Data Canopy can support tens of thousands of bivariate statistics.

conducted with the U workload. In this experiment, Data Canopy operates in the online mode, i.e., Data Canopy is built as queries are executed.

Figure 4.18 shows both the maximum memory footprint of Data Canopy in each scenario and the memory footprint after executing 2000 queries. We report the memory footprint of both univariate and bivariate statistics. In the case of univariate statistics, the maximum memory footprint is 1GB, and under memory pressure, it can incrementally shrink down to just 10MB. The maximum memory footprint of bivariate statistics is 32GB and, in a similar fashion, can shrink down to just 490MB. More generally, Data Canopy is able to vary its overall size (by changing its chunk size) to fit within the available main memory. Overall, the usage of the U workload remains less than one-third of the maximum size.

Next, we show that under memory pressure Data Canopy can still efficiently support tens of thousands of bivariate statistics over a wide range of data sizes. In this analysis, the main memory budget is set to 16GB, and Data Canopy operates in Phase 2 of the out-of-memory policy. The chunk size is equal to a page size (64KB). All univariate segment trees are in memory. Figure 4.19 shows the number of bivariate segment trees that Data Canopy supports in the remaining amount of main memory across a wide range of data sizes. Each of the segment trees can be used to answer a bivariate statistic over any range of a pair of columns.

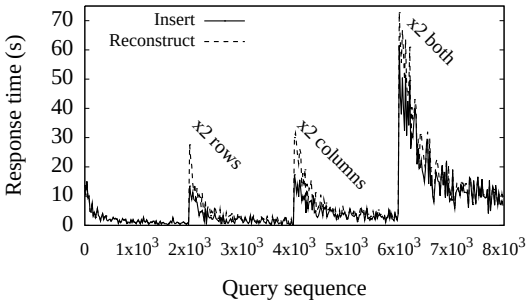


Figure 4.20: Data Canopy gracefully handles new data.

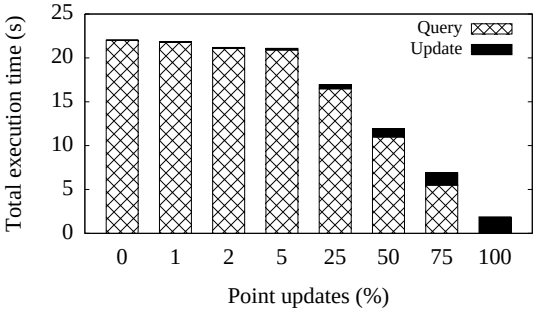


Figure 4.21: Updates in Data Canopy result in negligible overhead.

We observe that even for large data sets (1T rows and 1000 columns, data size to memory ratio of 1:250), Data Canopy can still efficiently support up to 10000 bivariate statistics, in addition to all univariate statistics.

4.10.6 Insertions and Updates

Now we show that Data Canopy seamlessly handles insertions of new data as well as updates to existing data.

Insertions. First, we show that Data Canopy efficiently handles insertions of new rows and new columns. We compare how Data Canopy incrementally handles updates to a strategy, where Data Canopy is built anew every time new data is added. We call this the reconstruct strategy. In this experiment, Data Canopy starts off with 25 columns and 100 million rows and operates in the online mode. New data is added in three phases: (1) the number of rows are doubled, (2) the number of columns are doubled, and (3) both the number of rows and columns are doubled. There is an interval of 2000 queries between each of the phases. At any point in time, we run the U workload that targets all data that is in the system. We report the response time as the sequence of queries passes through the three phases in Figure 4.20. We observe that as new data is added, there is an initial increase in response time that converges to the optimal for both strategies. The incremental strategy employed by Data Canopy results in lower initial overhead as well as converges faster to stable performance as compared to the

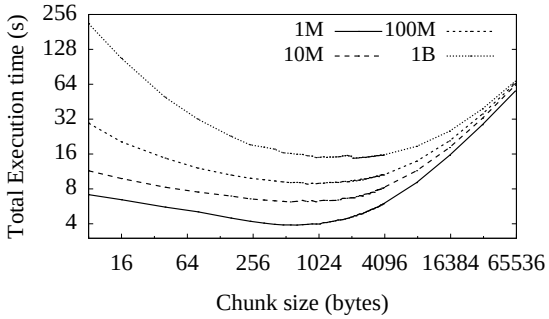


Figure 4.22: Data Canopy's query performance is a convex function of its chunk size.

reconstruct strategy. This is because in the incremental strategy, both the insertion of new rows and new columns is handled in a lightweight manner (merely adding metadata to the catalog) and basic aggregates are materialized only when and if queries target the new data. In addition to this, the existing library of basic aggregates is completely reused, whereas with the reconstruct strategy, the library is built from scratch after every insertion phase.

Updates. Next, we show that response time is minimally impacted in the presence of updates to existing data. We show this for a varying percentage of updates in the workload. We set up an experiment with 100 columns and 100 million rows. We run 2000 queries from the U workload with varying percent of point updates in the workload. Figure 4.21 shows the total execution time as we increase the proportion (percentage) of point updates in the workload. As we increase the proportion of point updates in the workload, the number of read queries decreases resulting in an overall decrease in execution time. Throughout this time, the overhead introduced by point updates remains low. In low updates scenarios (1 to 5 percent point updates) the overhead is less than 1 percent. For extremely high update scenarios (25 to 75 percent point updates), the average update overhead remains below 10 percent of execution time.

4.10.7 Model Verification

We now verify the query cost model that we developed in Section 4.6. Similar to the analysis in Figure 4.7, we vary the chunk size for various number of rows and observe

how this affects performance. The results are shown in Figure 4.22. We report the total execution time of 10K queries from the U workload on 100 columns.

There are two observations. First, the experimental results verify the behavior we see from the model in Figure 4.7. That is, there is a convex shape and for all data sizes there is a common chunk size area where we get the optimal overall performance. Second, this area is actually quite large (the x-axis is logarithmic) and so picking any chunk size that is close enough to the center of this area gives optimal behavior. A positive side-effect of this is that we do not have to make our query cost model any more complex, i.e., by adding separate weights for when a cache access is a miss or a hit to capture the different latencies (traversing a segment tree will typically cause cache misses while scanning chunks at the end-points of the query range (residual range) will typically cause a cache miss followed by more than one cache hits). Capturing simply the number of accessed cache lines allows us to get an estimate close enough in the optimal range, i.e., our analysis (as shown in Figure 4.7) estimates the optimal chunk size to be 220 bytes while Figure 4.22 shows that indeed 220 bytes is within the optimal range. An important side-effect of taking advantage of this behavior is that we do not need a training process for different machines (e.g., to figure out the cost of different accesses) - all we need is the cache line size. To fully optimize performance we pick a chunk size that is a multiple of the cache line. That is, the model gives us an optimal chunk size of 220 bytes, which we translate to a default chunk size of 256 bytes (4 times 64 which is the cache line size).

5

Deep Collider: Enabling Better Neural Network Design

We now present Deep Collider in detail. Deep Collider enables better deep learning model design by demystifying the relationship between a neural network model and various metrics of interest. The fundamental question we address with Deep Collider is given a number of parameters, what deep learning model to use. Deep Collider answers this question holistically, considering not only metrics related to the quality of the model (such as accuracy) but also those that have to do with the cost of training and deploying such models. Deep Collider targets a part of the deep learning model design space that is not very well understood: How to decide between single network models and ensembles of models under a parameter budget. Deep Collider establishes that deep neural networks' ensembles are more useful than single models for a more comprehensive range of use cases than previously understood. It also provides various ensemble design guidelines to optimize for both the quality and the cost of ensembles. This chapter describes the Deep Collider framework, the design space it analyzes, and the various design guidelines it uncovers.

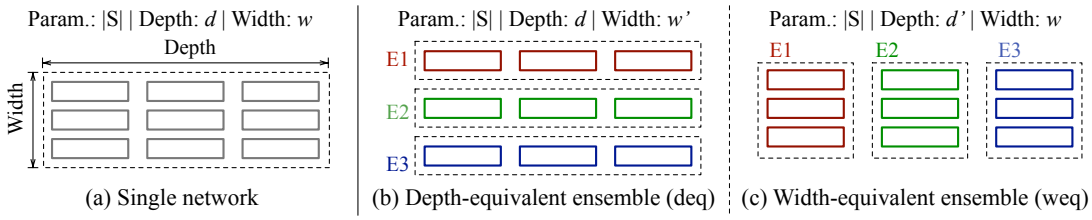


Figure 5.1: We explore a design space consisting of three design classes: (a) Single convolutional network models, (b) Depth-equivalent ensembles, and (c) Width-equivalent ensembles. The two ensemble design classes are created by distributing either the width factor or the depth corresponding to the single network amongst the ensemble networks while keeping the other factor fixed.

5.1 Framework: Design Space

The design space we explore consists of single convolutional neural network models and two classes of architecturally-homogenous ensembles. These ensemble classes help isolate the effect of the two design knobs – depth and width – on the quality and cost of an ensemble design. We first describe how we ensure a robust comparison of alternative model designs and then explain the degrees of freedom we explore.

Establishing Grounds for Fair Ranking. A key element of our framework is that the possible model designs are compared to each other only under equivalent resources. We ensure this by only comparing designs that have the same number of parameters. This comparison allows us to separate the quality of a design from the amount of resources given to it. Another way to think about this is that given a parameter budget, we can investigate how the three design classes rank for all relevant metrics (training and inference time, accuracy, and memory usage).

We fix the number of parameters because of its two distinctive properties over other metrics (that we could have fixed), such as training time, inference time, or accuracy: First, the number of parameters of a network is directly proportional to all other resource-related metrics. Second, the number of parameters is independent of the hardware or the software platform used and can be computed exactly from a network specification.

The Single Network Versus Ensemble Design Space. Our design space considers a convolutional neural network architecture $S^{(w,d)}$ from a class of neural network architectures C. $S^{(w,d)}$ has width factor w, depth d, and number of parameters $|S|$. Similarly an ensemble is described as $E = \{E_1 \dots E_k\}$. Ensembles are architecturally-homogenous i.e., all ensemble networks $E_1 \dots E_k$ have the same architecture and each network has $|E|/k$ parameters. When we compare a single network $S^{(w,d)}$ from C with an ensemble E we ensure that $E_1 \dots E_k \in C$ and $|E_1| + \dots + |E_k| = |S|$.

The reason why we restrict the design space to homogenous ensembles is to reduce the otherwise intractably large space[1] of all possible ensembles given a single network to a size that we can feasibly and thoroughly experiment with and reason about. Furthermore, many neural network ensembles introduced in research and used in practice are similarly homogenous, for instance, SnapShot Ensembles and Fast Geometric Ensembles (Huang et al. 2017a; Garipov et al. 2018). Additionally, our method provides a deterministic procedure of going between single network models and ensembles given a certain amount of parameters. Major sources of diversity in neural network ensembles are random weight initialization and stochastic training, both of which we incorporate in our framework.

Depth-Equivalent and Width-Equivalent Ensembles. Convolutional neural network architectures are determined by two design knobs – the depth and the width factor. Corresponding to these two design knobs, we create two classes of ensembles: depth-equivalent ensembles and width-equivalent ensembles. These are depicted in Figure 5.1: In depth-equivalent ensembles, the depth of the individual ensemble networks is the same as S (i.e., d), and the width factor is set to the highest possible value (i.e., w') without exceeding the parameter budget of $|S|$. In width-equivalent ensembles, on the other hand, the width factor is conserved across all ensemble networks (i.e., w), and the depth is modulated to the highest possible value (i.e., d') without exceeding $|S|$:

$$w' : k \cdot |E_i^{(w',d)}| \leq |S^{(w,d)}| \leq k \cdot |E_i^{(w'+1,d)}| \qquad d' : k \cdot |E_i^{(w,d')}| \leq |S^{(w,d)}| \leq k \cdot |E_i^{(w,d'+1)}|$$

[1] Given a single network with $|S|$ number of parameters, there are $\left\{ {|S| \atop k} \right\}$ (Stirling number of the second kind) as many ways of forming ensembles of size k. This number grows at a similar rate to exponential polynomials, e.g., $\left\{ {100 \atop 4} \right\} \approx 10^{59}$.

Architecture	Data sets	Epochs	Lr schedule	batch size
DenseNet	SVHN	100	0.1, 0.01(30), 0.001(60)	128
DenseNet	C10 and C100	120	0.1, 0.01(60), 0.001(90)	128
DenseNet	Tiny ImageNet	90	0.1, 0.01(30), 0.001(60)	64
DenseNet	ImageNet32-1K	90	0.1, 0.01(30), 0.001(60)	64
ResNet	C10	250	0.1, 0.01(100), 0.001(200)	128
ResNet	C100	500	0.1, 0.01(250), 0.001(375)	128
Wide ResNet	C10 and C100	200	0.1, 0.01(100), 0.001(200)	128

Table 5.1: For all networks, we use training hyperparameters listed in their respective papers (lr: learning rate).

The above definition follows that neural networks in depth-equivalent ensembles have higher depth than those in width-equivalent ensembles. Width-equivalent ensembles contain wider neural networks than their depth-equivalent counterparts. In this way, we isolate and study the effect of depth and width on ensemble accuracy as well as the resource requirement.

Overall, our design space spans three classes of convolutional neural network designs: (i) single network models, (ii) width-equivalent ensembles, and (iii) depth-equivalent ensembles. Every class contains several model designs instantiated by the four-tuple $\{w, d, |S|, C\}$. We next describe how we designed an exhaustive experimental framework to cover various configurations of these four-tuples.

5.2 Framework: Data, Architectures, and Metrics

Datasets and Architectures. We include widely-used state-of-the-art network architectures and data sets in our study. These include DenseNets, and (Wide) ResNets architectures as well as SVHN, C10 and C100, and ImageNet datasets. Table 5.1 summarizes these networks and training data sets as well as corresponding hyperparameters. We implement our experimental framework in PyTorch and used an Nvidia V100 GPU to run all experiments.

Evaluation Metrics. We study all three design classes – single network, width- and depth-equivalent ensembles – across five metrics: (i) generalization accuracy, (ii) train-

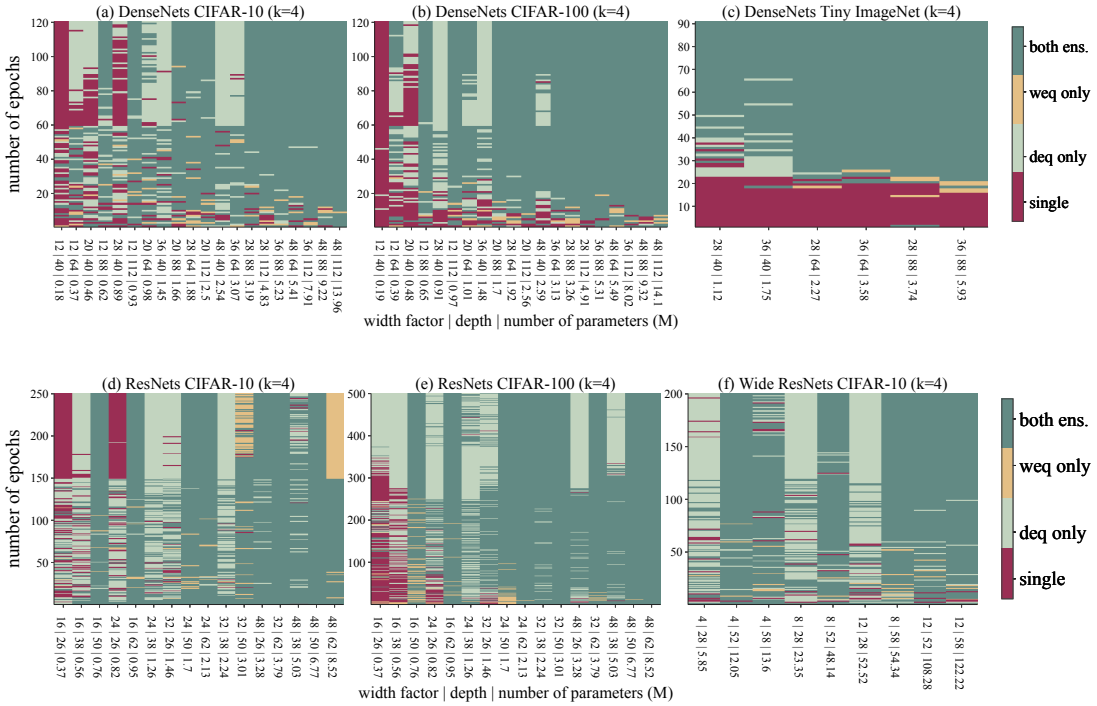


Figure 5.2: The Ensemble Switchover Threshold (EST) occurs consistently across various network architectures and data sets. Beyond this resource threshold, ensemble designs outperform single network models.

ing time per epoch, (iii) time to accuracy, (iv) inference time, and (v) memory usage. When considered together, these metrics provide a holistic picture of the quality and practicality of models.

5.3 Guideline: Ensembles Outperform Single Network Models After a Low to Moderate Parameter Threshold

We observe that both classes of ensembles – depth- and width-equivalent – outperform single network models after a resource threshold. We call this threshold the Ensemble Switchover Threshold (EST). Beyond the EST, ensemble models achieve 1 to 3 percent lower test error rates (across various architectures and data sets) compared with single

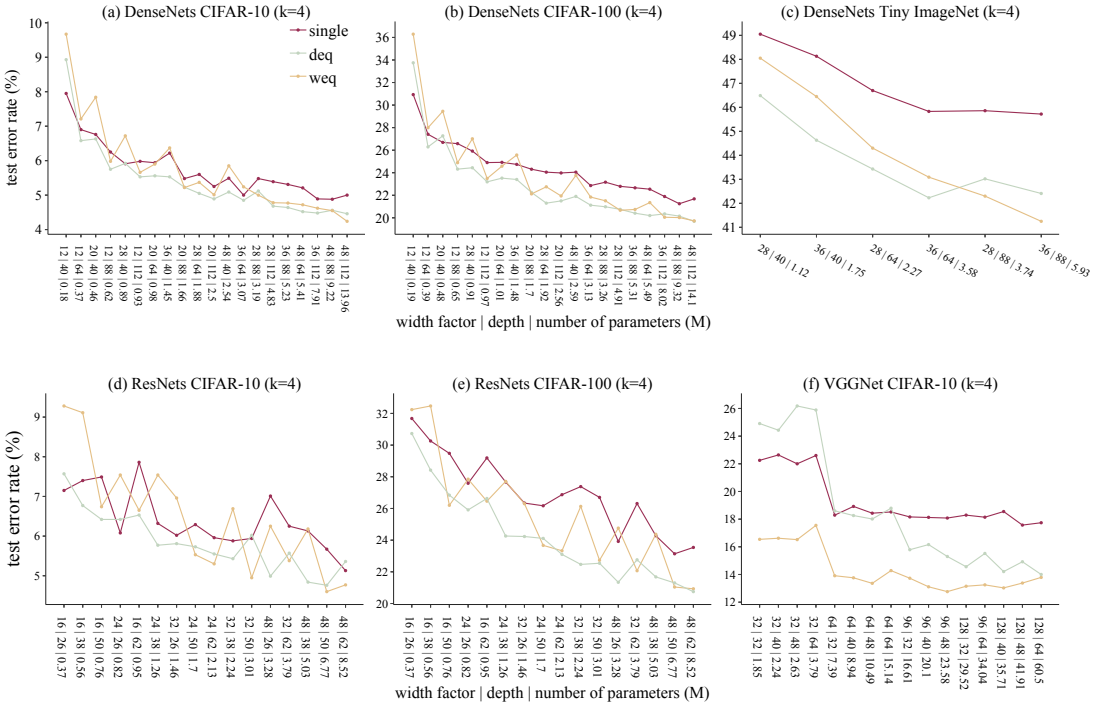


Figure 5.3: Ensembles arrive at lower test error rates than single network models after the EST has been reached.

networks having the same number of parameters.

The EST appears consistently across a wide range of data sets and architectures (Figure 5.2(a) through Figure 5.2(f)) as well as ensemble sizes (Figure 5.4(a) through Figure 5.4(c) and Figure 5.5(a) through Figure 5.5(c)). In these figures, we use discrete heat maps to visualize which of the three design classes – single network models (single), depth-equivalent ensembles (deq), and width-equivalent ensembles (weq) – dominates in terms of generalization accuracy for a given resource budget. This resource budget takes the form of the number of parameters (on the x-axis) and epochs (on the y-axis). We also mark areas where both classes of ensembles outperform single network models. Figure 5.3 shows the test error rates achieved on various data sets for DenseNet models.

The occurrence of EST both expands and questions the general consensus on the relative effectiveness of ensemble versus single network models. First, even when allocated the

same amount of resources, ensemble models still outperform single network models. This observation expands upon past empirical studies that only show how a k-network ensemble is more accurate than any of the single network models that it contains (Lee et al. 2015a; Huang et al. 2017a). Second, the EST occurs in low- to moderate-resource settings. For instance, in all of our experiments, we observe the EST at the 1M to 1.5M parameter range[2] and after no later than half of the training epochs. This trend challenges the widespread notion that neural network ensembles are useful only when we have tons of resources at our disposal (Lee et al. 2015a; Ju et al. 2017). Overall, our results indicate that *ensembles of convolutional models are preferable to single network models for a much wider range of use cases than previously understood.*

On the Superior Generalization of Ensembles Under a Parameter Budget. To interpret why ensembles outperform single network models under a parameter budget, we use the phenomenon of diminishing returns on increasing model sizes. In the past, this effect has been independently investigated by (Eigen et al. 2013) and (Dauphin and Bengio 2013) for a single network model. We hypothesize that as we increase the number of parameters, single network models exhibit more diminishing returns and plateau faster than ensemble networks. When the single network's generalization accuracy starts showing diminishing returns, the corresponding width-equivalent and depth-equivalent ensembles have smaller networks with $1/k$ as many parameters (assuming the parameters are spread equally along k ensemble networks). These individual networks in the ensemble are affected less by the plateau because they have $1/k$ as many parameters as the single network model. Thus, utilizing these networks in an ensemble leads to better generalization accuracy overall because they do not hit the threshold of the diminishing returns while still being able to benefit from the known properties that ensembles provide: (i) They enrich the space of hypotheses that are considered by the base model class and (ii) By averaging over various models, ensembles reduce the variance of base models, smoothing out variations due to initialization and the learning process.

[2]Wide ResNets are an exception: This is because, compared to other convolutional architectures, Wide ResNets have an order of magnitude more parameters even for modest depths and width factors. For instance, networks presented in the Wide ResNet paper have anywhere between 8M to 37M parameters compared to the 1M to 10M range for DenseNets (Zagoruyko and Komodakis 2016).

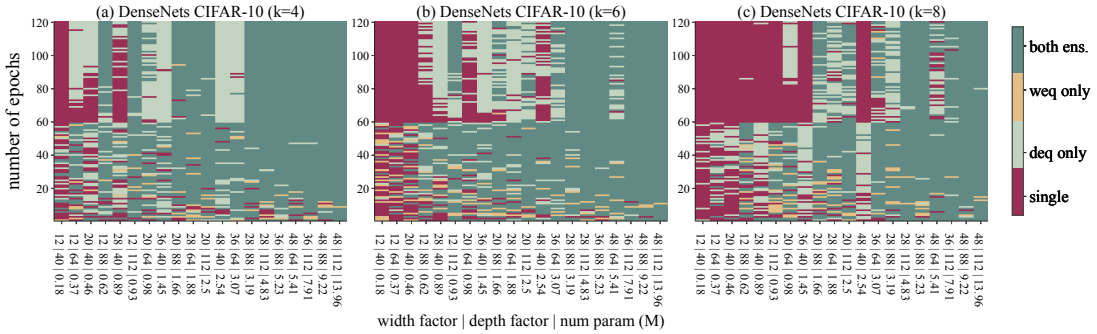


Figure 5.4: The Ensemble Switchover Threshold moves to the right as we increase the number of networks in the ensemble.

Next, we parse out how the data complexity and the composition of the ensemble networks affect the EST and, in turn, the ranking of the three design classes.

Ensembles are Even More Effective for More Complex Data Sets. We observe that the EST shifts closer to the origin as the training data set's complexity increases. This can be seen in Figure 5.2(a) through 5.2(c) where we train DenseNets on progressively more complex data sets (CIFAR-10, CIFAR-100, and Tiny ImageNet). This observation indicates that ensemble models are preferable to single models when training on more complex data sets for an even wider range of available resources. This observation again expands the utility of ensembles. There is theoretical and empirical work establishing that ensembles do better for complex data (Bonab and Can 2017; Huang et al. 2017a). We, however, establish this phenomenon in the resource-equivalent setting as opposed to past studies that do so for ensembles and single networks with drastically different numbers of parameters.

Large Ensembles are Effective Under a Large Parameter Budget. As we increase the number of networks k within an ensemble without increasing the parameter budget, the overall accuracy of ensemble designs diminishes, pushing the EST to a higher resource limit. Figure 5.4 demonstrates this phenomenon for DenseNets and Figure 5.5 shows it for ResNets. For instance, for DenseNet models, the EST moves from the 1.5M range for $k = 4$ to the 3M range for $k = 6$ and, then, to the 5M range for $k = 8$. This

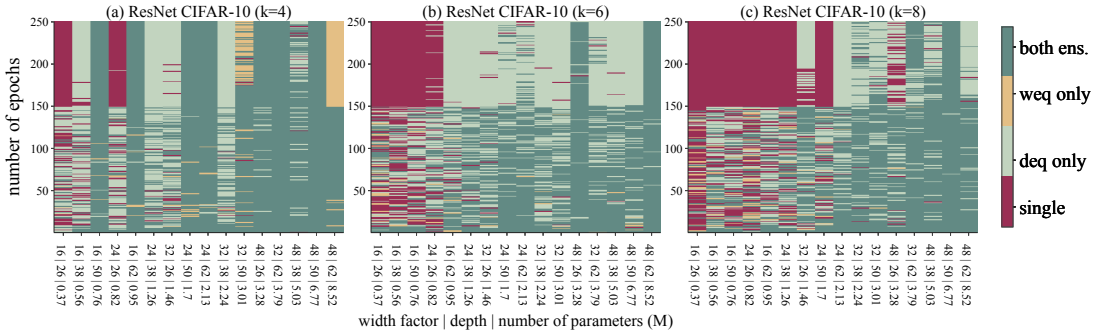


Figure 5.5: The Ensemble Switchover Threshold moves to the right as we increase the number of networks in the ensemble. Here, we demonstrate this phenomenon for ResNet models.

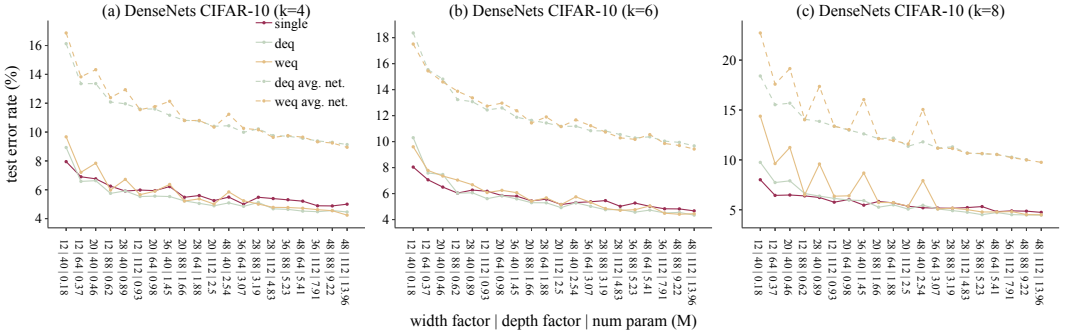


Figure 5.6: As we increase the size of ensembles, accuracy of individual networks in the ensemble decreases. This results in an overall reduction in ensemble accuracy shifting the EST to the high-resource space.

shift can be explained by looking at individual accuracy of ensemble networks. Figure 5.6 shows test error rates of the ensemble as a whole as well as the average test error rates of individual ensemble networks corresponding to Figure 5.4. We observe that as we increase the number of networks k (from $k = 4$ in Figure 5.6(a) to $k = 8$ in Figure 5.6(c)), the individual test error rates (shown as dotted lines) increases. This increase happens because individual networks' size goes down (as we keep a fixed parameter budget). This observation implies that larger-size ensembles are desirable over smaller sizes only when we have a sufficient parameter budget to assign to every single network

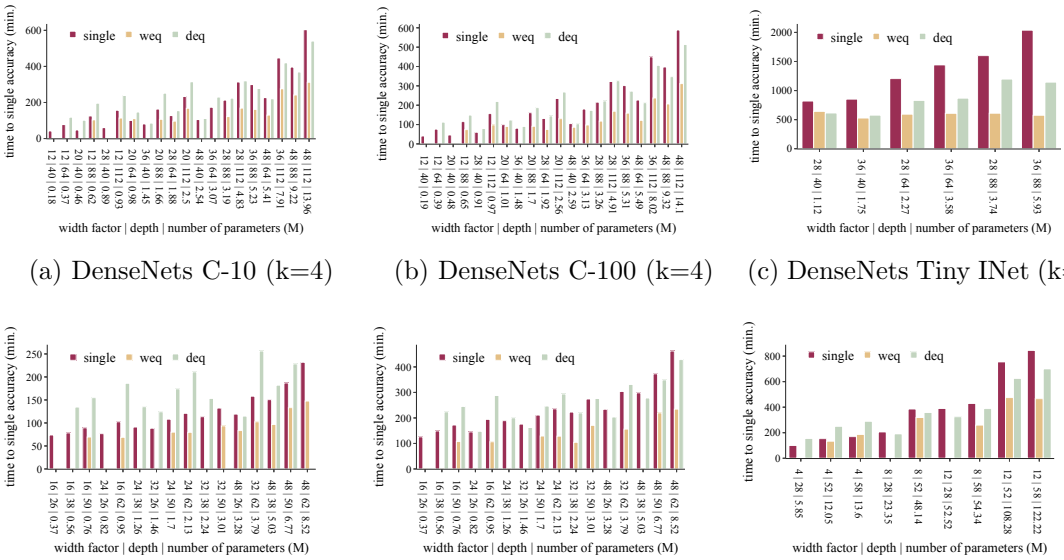
in the model. As opposed to previous work, our experiments decouple the parameter budget from the number of networks in the model. We discover that just increasing the ensemble size without increasing the total number of parameters hurts accuracy.

Depth-Equivalent Ensembles Outperform Width-Equivalent Ensembles. We observe that depth-equivalent ensembles are overall more accurate than width-equivalent ensembles (as shown in Figure 5.3). They also consistently demonstrate EST at a lower resource range. This can be explained by the fact that modern convolutional neural network architectures provide better accuracy with increasing depth (Eigen et al. 2013; Urban et al. 2016). Here, depth-equivalent ensembles have deeper ensemble networks with better individual accuracy. Thus, when used together in an ensemble, they also provide better ensemble accuracy. This way, when designing ensemble models for high accuracy, deeper networks are preferable to wider networks.

EST vs. Memory Split Advantage. For a limited part of the design space, recent work has observed the existence of a parameter limit beyond which depth-equivalent ensembles outperform single networks. This is termed as the Memory Split Advantage or the MSA (Kondratyuk et al. 2020). The EST, however, is not defined just with respect to the number of parameters but also the number of training epochs, inference cost, and memory usage as by only looking at these metrics in conjunction, we can get a complete picture of the relative effectiveness of ensembles vs. single networks. In this way, the EST subsumes the MSA, and we also verify the MSA across a significantly larger design space (e.g., we consider 3× more data sets and twice as many architecture types) than has been done before.

On the Instability of Width-Equivalent Ensembles. The performance of width-equivalent ensembles (weq ensembles) exhibits unstable behavior. In particular, it has local spikes when it comes to the test error rates. This is particularly pronounced for the ResNet ensembles (Figure 5.3(d) and Figure 5.3(e)).

Our interpretation of this phenomenon is that these local spikes have to do with the relative depth of networks in the width-equivalent ensemble designs. When comparing designs that are close together in the parameter range, we observe that the designs with more depth (of ensemble networks) generally outperform those with less depth even if



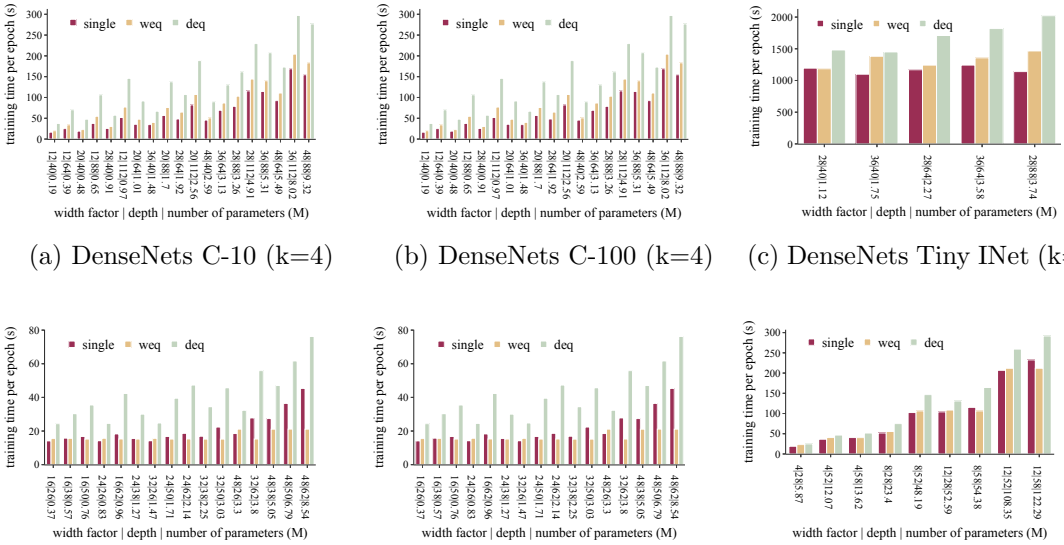
(a) DenseNets C-10 (k=4) (b) DenseNets C-100 (k=4) (c) DenseNets Tiny INet (k=4)

(d) ResNets C-10 (k=4) (e) ResNets C-100 (k=4) (f) Wide ResNets C-10 (k=4)

Figure 5.7: When ensemble designs can provide better accuracy, they can also do so faster than single network models (missing bars indicate that designs cannot reach single network model accuracy).

the latter have more parameters. The depth of networks in the weq ensemble plays a more dominant role than the total number of parameters they have. As an example of this, consider Figure A(a) and A(b) in the revised version of the paper (ResNet CIFAR-10 (k=4) and ResNet CIFAR-100 (k=4). Weq exhibit three spikes at 2.24M, 3.28M, and 5.03M parameters. All three of these points are flanked on both sides by designs that have similar number of parameters but more depth.

This observation is consistent with past observations that depth is more influential in determining the accuracy of networks (Eigen et al. 2013).



(a) DenseNets C-10 (k=4) (b) DenseNets C-100 (k=4) (c) DenseNets Tiny INet (k=4)

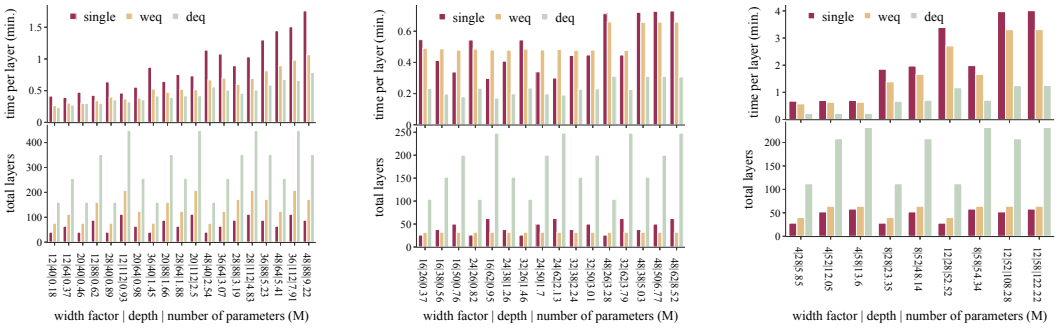
(d) ResNets C-10 (k=4) (e) ResNets C-100 (f) Wide ResNets C-10 (k=4)

Figure 5.8: Depth-equivalent ensembles take longer to train per epoch as compared to single network models. Width-equivalent ensembles, on the other hand, take comparable time.

5.4 Guideline: Ensembles Train Faster and Provide Similar Inference Time

First, we analyze the training time. Despite taking longer per epoch, both ensemble classes achieve the accuracy of single network models significantly faster for a considerable part of the design space (e.g., 1.2× to 5× faster across our experiments). This happens after the EST has been reached, i.e., *when ensemble designs can provide better accuracy, they can also do so faster than single network models.* This can be seen in Figure 5.7. Here, we plot the total training time needed for any of the three design classes to achieve the maximum accuracy of single network models under the same parameter budget. Figure 5.8 shows the corresponding training time per epoch.

The Combined Depth Determines Per Epoch Training Time.. We observe that both classes of ensembles, on average, take longer to train per epoch as compared to

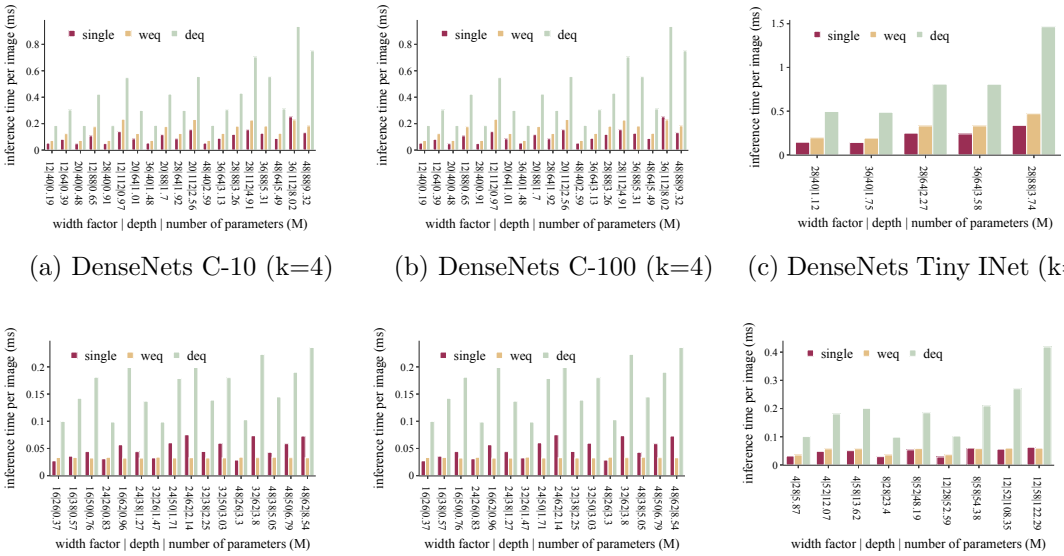


(a) DenseNets on CIFAR-10 (b) ResNets on CIFAR-10 (c) Wide ResNets on CIFAR-10

Figure 5.9: We break down per epoch training time into: (i) time spent per layer and (ii) total number of layers. We observe that the total number of layers in the model more significantly determines the per epoch training time as compared to the width. The ensemble size is 4 across all these experiments.

single network models as they train k networks instead of one. How much more time ensembles take per epoch depends heavily on the ensemble networks' design: This additional time is negligible for width-equivalent ensembles whereas, for depth-equivalent ensembles, it results in $2\times$ more expensive per epoch training. This trend can be explained by how the training time per epoch scales with respect to the width and depth of convolutional neural network models. This ultimately connects to how GPUs process data, which is more favorable for networks with few wider layers as compared to those that have several thinner layers.

We break down the training time per epoch of all designs into two constituents: time spent per layer and number of layers. Figure 5.9shows this breakdown for various architectures. We observe that the total number of layers in a model (for ensembles, this is the sum of all networks' depth) majorly determines the training time per epoch. For the same parameter budget, depth-equivalent ensembles have proportionally more layers, whereas width-equivalent ensembles have proportionally more width. The average time per layer depends on the width and does not increase significantly as we move from depth-equivalent ensembles to width-equivalent ensembles. On the other hand, the total number of layers scales linearly with depth. For the same parameter budget, the total number of layers is significantly higher for depth-equivalent ensembles than



(a) DenseNets C-10 (k=4) (b) DenseNets C-100 (k=4) (c) DenseNets Tiny INet (k=4)

(d) ResNets C-10 (k=4) (e) ResNets C-100 (k=4) (f) Wide ResNets C-10 (k=4)

Figure 5.10: Width-equivalent ensembles take comparable time to single network models for inference. Depth-equivalent ensembles take significantly longer.

the other two designs, resulting in higher per epoch training.

From a GPU perspective, wider and shallower networks are more efficient to execute than narrower and deeper networks for the same parameter budget. This can be attributed to the massive amount of data parallelism in modern GPUs. Increasing the network's width just increases the number of kernels within layers. This increase more efficiently utilizes GPU's massive capacity to perform the same operation on multiple data items. On the other hand, deepening a network introduces new layers (and operations) that require additional synchronization steps slowing down the overall execution.

Networks in Ensemble Models Converge Faster than Single Network Models for the Same Parameter Budget. The fact that ensemble designs can reach the same accuracy faster than single network models can be attributed to the fact that, for the same parameter budget, all networks in the ensemble model are smaller than the single network model. Smaller networks are known to converge faster albeit to lower accuracy than larger networks.

However, we observe that the distinct advantage ensemble designs provide over the single model is that when we use smaller networks in an ensemble, we get the best of both worlds. We converge faster at an individual network level, and ensembling makes up for the generalization accuracy.

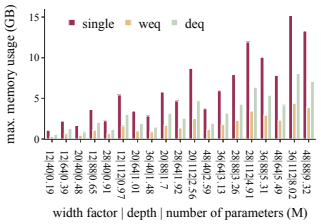
Overall, these observations again question the conventional wisdom of ensembles being significantly slower to train than single network models. When we analyze the design space under a fixed parameter budget, we uncover that for a vast range of the design space: (i) width-equivalent ensembles introduce negligible overhead to per epoch training time as compared to single network models and (ii) both ensemble designs achieve and surpass accuracy of single network models in considerably less training time.

Width-Equivalent Ensembles Provide Competitive Inference Time. We provide the inference time per image in Figure 5.10 and observe a similar trend to training time per epoch. While depth-equivalent ensembles are significantly slower, *width-equivalent ensembles provide comparable inference speed to single network models.* Again, this questions conventional wisdom that expects ensembles to be substantially slower in inference.

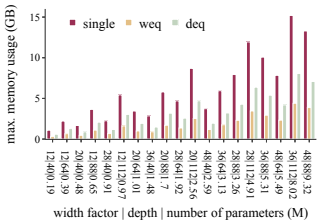
5.5 Guideline: Ensembles are Memory Efficient

Regarding memory usage we observe that the trend favors both classes of ensemble designs over single network models. Figure 5.11 provides the amount of memory used as we train depth-equivalent ensembles, width-equivalent ensembles, and single network models. This is the minimum amount of memory that a GPU needs to train any of these designs for the batch sizes provided in Table 5.1. This memory is majorly used to store model parameters and intermediate results.

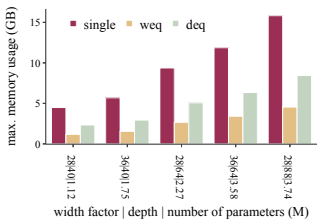
The superior memory efficiency of ensemble models is because when we train a k-networks ensemble, at any point during the training process, we only need as much memory to train one of the k networks (having $\frac{1}{k}$ as many parameters compared to the single network). This observation has two important implications: First, we can use larger batch sizes for the same GPU while training an ensemble of networks. This, for



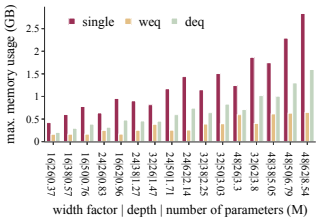
(a) DenseNets C-10 (k=4)



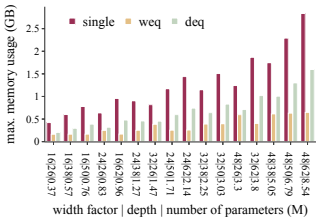
(b) DenseNets C-10 (k=4)



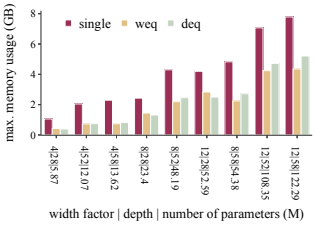
(c) DenseNets Tiny INet (k=4)



(d) ResNets C-10 (k=4)



(e) ResNets C-100 (k=4)



(f) Wide ResNets C-10 (k=4)

Figure 5.11: Both classes of ensemble models are significantly more memory efficient.

instance, is useful when training complex data sets such as ImageNet. Additionally, we can feasibly train the same number of parameters in an ensemble using lower-end GPUs with less memory.

6

MotherNets: Rapid Deep Ensemble Learning

We now present MotherNets in detail. MotherNets enable higher accuracy and practical training cost for large and diverse neural network ensembles. A MotherNet captures the structural similarity across some or all members of a deep neural network ensemble, allowing us to share data movement and computation costs across these networks. We first train a single or a small set of MotherNets, and, subsequently, we generate the target ensemble networks by transferring the function from the trained MotherNet(s). Then, we continue to train these ensemble networks, which now converge drastically faster than training from scratch. In this chapter, we describe how to train ensembles through MotherNets and experimentally analyze how MotherNets establishes a Pareto-frontier for the accuracy-training time tradeoff of ensemble networks.

6.1 Constructing MotherNets

Definition: MotherNet. Given a cluster of k neural networks $C = \{N_1, N_2, \ldots N_k\}$, where N_i denotes the i-th neural network in C, the MotherNet M_c is defined as the largest network from which all networks in C can be obtained through function-preserving

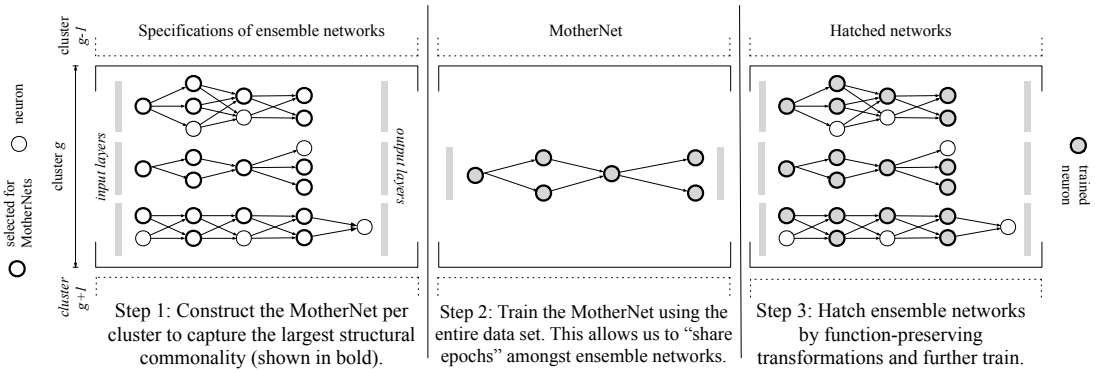


Figure 6.1: MotherNets train an ensemble of neural networks by first training a set of MotherNets and transferring the function to the ensemble networks. The ensemble networks are then further trained converging significantly faster than training individually.

transformations. MotherNets divide an ensemble into one or more such network clusters and construct a separate MotherNet for each.

Constructing a MotherNet for Fully-Connected Networks. Assume a cluster C of fully-connected neural networks. The input and the output layers of M_c have the same structure as all networks in C, since they are all trained for the same task. M_c is initialized with as many hidden layers as the shallowest network in C. Then, we construct the hidden layers of M_c one-by-one going from the input to the output layer. The structure of the i-th hidden layer of M_c is the same as the i-th hidden layer of the network in C with the least number of parameters at the i-th layer. Figure 6.1 shows an example of how this process works for a toy ensemble of two three-layered and one four-layered neural networks. Here, the MotherNet is constructed with three layers. Every layer has the same structure as the layer with the least number of parameters at that position (shown in bold in Figure 6.1 Step (1)).

Algorithm 6.1 describes how to construct the MotherNet for a cluster of fully-connected neural networks. We proceed layer-by-layer selecting the layer with the least number of parameters at every position.

Constructing a MotherNet for Convolutional Networks. Convolutional neural network architectures consist of blocks of one or more convolutional layers separated

by pooling layers (He et al. 2016; Shazeer et al. 2017; Simonyan and Zisserman 2015; Szegedy et al. 2015). These blocks are then followed by another block of one or more fully-connected layers. For instance, VGGNets are composed of five blocks of convolutional layers separated by max-pooling layers, whereas, DenseNets consist of four blocks of densely connected convolutional layers. For convolutional networks, we construct the MotherNet M_c block-by-block instead of layer-by-layer. The intuition is that deeper or wider variants of such networks are created by adding or expanding layers within individual blocks instead of adding them all at the end of the network. For instance, VGG-C (with 16 convolutional layers) is obtained by adding one layer to each of the last three blocks of VGG-B (with 13 convolutional layers) (Simonyan and Zisserman 2015). To construct the MotherNet for every block, we select as many convolutional layers to include in the MotherNet as the network in C with the least number of layers in that block. Every layer within a block is constructed such that it has the least number of filters and the smallest filter size of any layer at the same position within that block. An example of this process is shown in Figure 6.2. Here, we construct a MotherNet for three convolutional neural networks block-by-block. For instance, in the first block, we include one convolutional layer in the MotherNet having the smallest filter width and the least number of filters (i.e., 3 and 32 respectively).

Algorithm 6.2 provides a detailed strategy to construct the MotherNet for a cluster of convolutional neural networks. We proceed block-by-block, where each block is composed of multiple convolutional layers. The MotherNet has as many blocks as the network with the least number of blocks. Then, for every block, we proceed layer-by-layer and construct the MotherNet layer at every position as follows: First, we compute the least number of convolutional filters and convolutional filter sizes at that position across all ensemble networks. Let these be F_{min} and S_{min} respectively. Then, in MotherNet, we include a convolutional layer with F_{min} filters of S_{min} size at that position.

Constructing MotherNets for Ensembles of Neural Networks with Different Sizes and Topologies. By construction, the overall size and topology (sequence of layer sizes) of a MotherNet is limited by the smallest network in its cluster. If we were to assign a single cluster to all networks in an ensemble that has a large difference in size and topology between the smallest and the largest networks, there will be a corre-

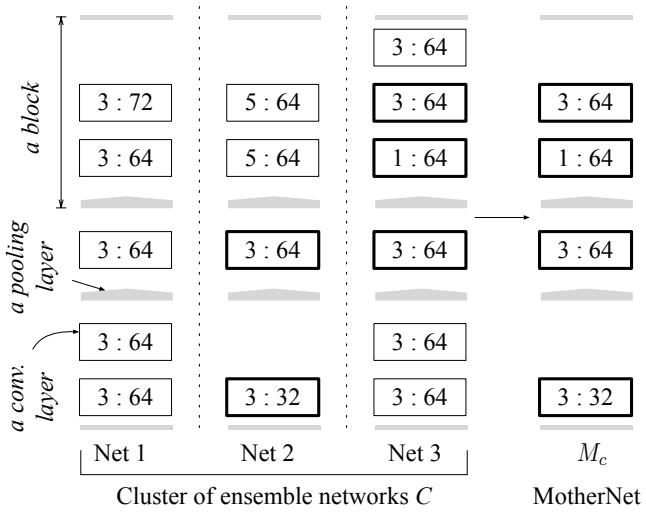


Figure 6.2: Constructing MotherNet for convolutional neural networks block-by-block. For each layer, we select the layer with the least number of parameters from the ensemble networks (shown in bold rectangles) (Notation: <filter_width> : <filter_number>).

spondingly large difference between at least one ensemble network and the MotherNet. This may lead to a scenario where the MotherNet only captures an insignificant amount of commonality. This would negatively affect performance as we would not be able to share significant computation and data movement costs across the ensemble networks. This property is directly correlated with the size of the MotherNet.

In order to maintain the ability to share costs in diverse ensembles, we partition such an ensemble into g clusters, and for every cluster, we construct and train a separate MotherNet. To perform this clustering, the m networks in the ensemble $E = \{N_1, N_2, \ldots N_m\}$ are represented as vectors $E_v = \{V_1, V_2, \ldots V_m\}$ such that V_i^j stores the size of the j-th layer in N_i. These vectors are zero-padded to a length of $\max(\{|N_1|, |N_2|, \ldots |N_m|\})$ (where $|N_i|$ is the number of layers in N_i). For convolutional neural networks, these vectors are created by first creating similarly zero-padded sub-vectors per block and then concatenating the sub-vectors to get the final vector. In this case, to fully represent convolutional layers, V_i^j stores a 2-tuple of filter sizes and number of filters.

Given a set of vectors E_v, we create g clusters using the balanced K-means algorithm while minimizing the Levenshtein distance between the vector representation of networks

Algorithm 6.1 Constructing the MotherNet for fully-connected neural networks

```
Input: E: ensemble networks in one cluster;
Initialize: M: empty MotherNet;

// set input/output layer sizes
M.input.num_param ← E[0].input.num_param;
M.output.num_param ← E[0].output.num_param;
M.num_layers ← getShallowestNetwork(E).num_layers;

// set hidden layer sizes
for i ← 0 ... M.num_layers-1 do
  | M.layers[i].num_param ← getMin(E,i);
end
return M;

// Get the min.  size layer at posn
Function getMin(E,posn)
  | min ← E[0].layers[posn].num_param;
  | for j ← 0 ... len(E)-1 do
  |   | if E[j].layers[posn].num_param < min then
  |   |   | min ← E[j].layers[posn].num_param
  |   | end
  | end
  | return min;
```

in a cluster and its MotherNet (Levenshtein 1966; MacQueen 1967). The Levenshtein or the edit distance between two vectors is the minimum number of edits – insertions, deletions, or substitutions – needed to transform one vector to another. By minimizing this distance, we ensure that, for every cluster, the ensemble networks can be obtained from their cluster's MotherNet with the minimal amount of edits constrained on g. During every iteration of the K-means algorithm, instead of computing centers of candidate clusters, we construct MotherNets corresponding to every cluster. Then, we use the edit distance between these MotherNets and all networks to perform cluster reassignments.

Constructing MotherNets for Ensembles of Diverse Architecture Classes. An individual MotherNet is built for a cluster of networks that belong to a single architecture class. Each architecture class has the property of function-preserving navigation. This is to say that given any member of this class, we can build another member of

Algorithm 6.2 Constructing the MotherNet for convolutional neural networks block-by-block.

```
Input: E: ensemble of convolutional networks in one cluster;
Initialize: M: empty MotherNet;

// set input/output layer sizes and number of blocks
M.input.num_param ← E[0].input.num_param;
M.output.num_param ← E[0].output.num_param;
M.num_blocks ← getShallowestNetwork(E).num_blocks;

// set hidden layers block-by-block
for k ← 0 ... M.num_blocks-1 do
    M.block[k].num_hidden ← getShallowestBlockAt(E,k).num_hidden; // select the
    shallowest block

    for i ← 0 ... M.block[k].num_hidden-1 do
        M.block[k].hidden[i]..num_filters,  M.block[k].hidden[i]..filter_size  ←  get-
        Min(E,k,i)
    end
end
return M;

// Get minimum number of filters and filter size at posn
Function getMin(E,blk,posn)
    min_num_filters ← E[0].block[blk].hidden[posn].num_filters;
    min_filter_size ← E[0].block[blk].hidden[posn].filter_size;
    for j ← 0 ... len(E) do
        if E[j].block[blk].hidden[posn].num_filters < min_num_filters then
            min_num_filters ← E[j].block[blk].hidden[posn].num_filters;
        end
        if E[j].block[blk].hidden[posn].filter_size < min_filter_size then
            min_filter_size ← E[j].block[blk].hidden[posn].filter_size;
        end
    end
    return min_num_filters, min_filter_size;
```

this class with more parameters but having the same function. Multiple types of neural networks fall under the same architecture class (Cai et al. 2018). For instance, we can build a single MotherNet for ensembles of AlexNets, VGGNets, and Inception Nets as well as one for DenseNets and ResNets. To handle scenarios when an ensemble con-

tains members from diverse architecture classes i.e., we cannot navigate the entire set of ensemble networks in a function-preserving manner, we build a separate MotherNet for each class (or a set of MotherNets if each class also has networks of diverse sizes).

6.2 Training MotherNets

Overall, the techniques described in the previous paragraphs allow us to create g MotherNets for an ensemble, being able to capture the structural similarity across diverse networks both in terms of architecture and topology. We now describe how to train an ensemble using one or more MotherNets to help share the data movement and computation costs amongst the target ensemble networks.

Training Step 1: Training the MotherNets. First, the MotherNet for every cluster is trained from scratch using the entire data set until convergence. This allows the MotherNet to learn a good core representation of the data. The MotherNet has fewer parameters than any of the networks in its cluster (by construction) and thus it takes less time per epoch to train than any of the cluster networks.

Training Step 2: Hatching Ensemble Networks. Once the MotherNet corresponding to a cluster is trained, the next step is to generate every cluster network through a sequence of function-preserving transformations that allow us to expand the size of any feed-forward neural network, while ensuring that the function (or mapping) it learned is preserved (Chen et al. 2016). We call this process *hatching* and there are two distinct approaches to achieve this: Net2Net increases the capacity of the given network by adding identity layers or by replicating existing weights (Chen et al. 2016). Network Morphism, on the other hand, derives sufficient and necessary conditions that when satisfied will extend the network while preserving its function and provides algorithms to solve for those conditions (Wei et al. 2016, 2017).

In MotherNets, we adopt the first approach i.e., Net2Net. Not only is it conceptually simpler but in our experiments we observe that it serves as a better starting point for further training of the expanded network as compared to Network Morphism. Overall, function-preserving transformations are readily applicable to a wide range of feed-

forward neural networks including VGGNets, ResNets, FractalNets, DenseNets, and Wide ResNets (Chen et al. 2016; Wei et al. 2016, 2017; Huang et al. 2017b). As such MotherNets is applicable to all of these different network architectures. In addition, designing function-preserving transformations is an active area of research and better transformation techniques may be incorporated in MotherNets as they become available.

Hatching is a computationally inexpensive process that takes negligible time compared to an epoch of training (Wei et al. 2016). This is because generating every network in a cluster through function preserving transformations requires at most a single pass on layers in its MotherNet.

Training Step 3: Training Hatched Networks. To explicitly add diversity to the hatched networks, we randomly perturb their parameters with gaussian noise before further training. This breaks symmetry after hatching and it is a standard technique to create diversity when training ensemble networks (Hinton et al. 2015; Lee et al. 2015b; Wei et al. 2016, 2017). Further, adding noise forces the hatched networks to be in a different part of the hypothesis space from their MotherNets.

The hatched ensemble networks are further trained converging significantly faster compared to training from scratch. This fast convergence is due to the fact that by initializing every ensemble network through its MotherNet, we placed it in a good position in the parameter space and we need to explore only for a relatively small region instead of the whole parameter space. We show that hatched networks typically converge in a very small number epochs.

We experimented with both full data and bagging to train hatched networks. We use full data because given the small number of epochs needed for the hatched networks, bagging does not offer any significant advantage in speed while it hurts accuracy.

Parallel Training. MotherNets create a new schedule for "sharing epochs" amongst networks of an ensemble but the actual process of training in every epoch remains unchanged. As such, state-of-the-art approaches for distributed training such as parameter-server (Dean et al. 2012) and asynchronous gradient descent (Gupta et al. 2016; Iandola et al. 2016) can be applied to fully utilize as many machines as available during any stage of MotherNets' training.

6.3 Navigating Accuracy-Training time Tradeoff

MotherNets can navigate the tradeoff between accuracy and training time by controlling the number of clusters g, which in turn controls how many MotherNets we have to train independently from scratch. For instance, on one extreme if g is set to m, then every network in E will be trained independently, yielding high accuracy at the cost of higher training time. On the other extreme, if g is set to one then, all ensemble networks have a shared ancestor and this process may yield networks that are not as diverse or accurate, however, the training time will be low.

MotherNets expose g as a tuning knob. As we show in our experimental analysis, MotherNets achieve a new Pareto frontier for the accuracy-training cost tradeoff which is a well-defined convex space. That is, with every step in increasing g (and consequently the number of independently trained MotherNets) accuracy does get better at the cost of some additional training time and vice versa. Conceptually this is shown in Figure **??**. This convex space allows robust and predictable navigation of the tradeoff. For example, unless one needs best accuracy or best training time (in which case the choice is simply the extreme values of g), they can start with a single MotherNet and keep adding MotherNets in small steps until the desired accuracy is achieved or the training time budget is exhausted. This process can further be fine-tuned using known approaches for hyperparameter tuning methods such as bayesian optimization, training on sampled data, or learning trajectory sampling (Goodfellow et al. 2016).

6.4 Shared-MotherNets: Enabling Fast Ensemble Inference

Shared-MotherNets. We introduce shared-MotherNets to reduce inference time and memory requirement of ensembles trained through MotherNets. In shared-MotherNets, after the process of hatching (step 2 from §6.2), the parameters originating from the MotherNet are incrementally trained in a shared manner. This yields a neural network ensemble with a single copy of MotherNet parameters reducing both inference time and memory requirement.

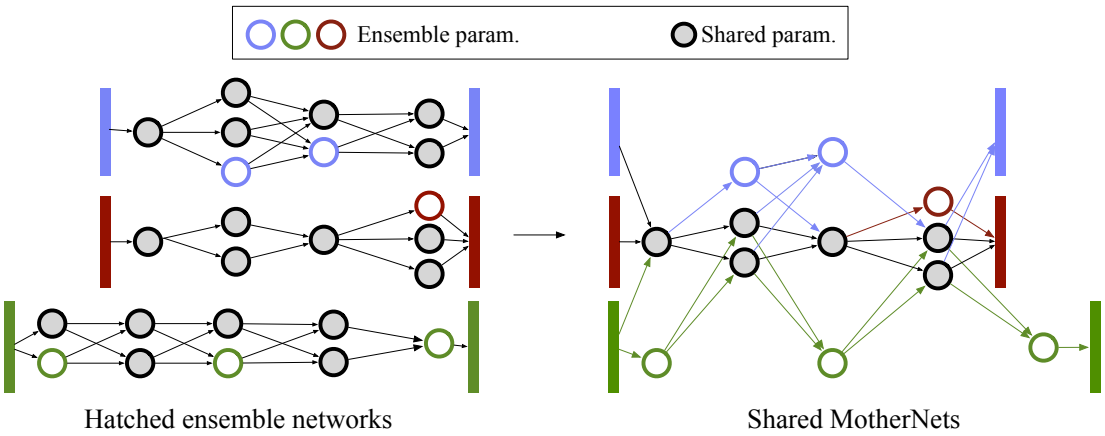


Figure 6.3: To construct a shared-MotherNet, parameters originating from the MotherNet are combined together in the ensemble.

Constructing a Shared-MotherNet. Given an ensemble E of K hatched networks (i.e., those networks that are obtained from a trained MotherNet), we construct a shared-MotherNet S as follows: First, S is initialized with K input and output layers, one for every hatched network. This allows S to produce as many as K predictions. Then, every hidden layer of S is constructed one-by-one going from the input to the output layer and consolidating all neurons across all of E that originate from the MotherNet. To consolidate a MotherNet neuron at layer l_i, we first reduce the k copies of that neuron (across all K networks in E) to a single copy. All inputs to the neuron that may originate from various other neurons in the layer l_{i-1} across different hatched networks are added together. The output of this consolidated neuron is then forwarded to all neurons in the next layer l_{i+1} (across all hatched networks) which were connected to the consolidated neuron.

Figure 6.3 shows an example of how this process works for a simple ensemble of three hatched networks. The filled circles represent neurons originating from the MotherNet and the colored circles represent neurons from ensemble networks. To construct the shared-MotherNet (shown on the right), we go layer-by-layer consolidating neurons originating from MotherNet.

The shared-MotherNet is then trained incrementally. This proceeds similarly to step 3 from §6.2, however, now through the shared-MotherNet, the neurons originating from

the MotherNet are trained jointly. This results in an ensemble that has K outputs, but some parameters between the networks are shared instead of being completely independent. This reduces the overall number of parameters, improving both the speed and the memory requirement of inference.

Memory Reduction. Assume an ensemble $E = \{N_0, N_1, \ldots N_{K-1}\}$ of K neural networks (where N_i denotes a neural network architecture in the ensemble with $|N_i|$ number of parameters) and its MotherNet M. The number of parameters in the ensemble is reduced by a factor of χ given by:

$$\chi = 1 - \frac{k|M|}{\sum_{i=0}^{K-1} |N_i|}$$

6.5 Experimental Analysis

We demonstrate that MotherNets enable a better training time-accuracy tradeoff than existing fast ensemble training approaches across multiple data sets and architectures. We also show that MotherNets make it more realistic to use large neural network ensembles i.e., those having dozens of neural networks.

Baselines. We compare against five state-of-the-art methods spanning both techniques that train all ensemble networks individually, i.e., Full Data (FD) and Bagging (BA), as well as approaches that generate ensembles by training a single network, i.e., Knowledge Distillation (KD), Snapshot Ensembles (SE), and TreeNets (TN).

Evaluation Metrics. We capture both the training cost and the resulting accuracy of an ensemble. For the training cost, we report the wall clock time as well as the monetary cost for training on the cloud. For ensemble test accuracy, we report the test error rate under the widely used ensemble-averaging method (Van der Laan et al. 2007; Guzman-Rivera et al. 2012, 2014; Lee et al. 2015b). Experiments with alternative inference methods (e.g., super learner and voting (Ju et al. 2017)) showed that the method we use does not affect the overall results in terms of comparing the algorithms.

Ens.	Member networks	Param.	SE alternative	Param.
V5	VGG 13, 16, 16A, 16B, and 19 from the VGGNet paper (Simonyan and Zisserman 2015)	682M	VGG-16 $\times$ 5	690M
D5	Two variants of DenseNet-40 (with 12 and 24 convolutional filters per layer) and three variants of DenseNet-100 (with 12, 16, and 24 filters per layer) (Huang et al. 2017b)	17M	Dense-60 $\times$ 5	17.3M
R10	Two variants each of ResNet 20, 32, 44, 56, and 110 from the ResNet paper (He et al. 2016)	327M	R-56 $\times$ 10	350M
V25	25 variants of VGG-16 with distinct architectures created by progressively varying one layer from VGG16 in one of three ways: (i) increasing the number of filters, (ii) increasing the filter size, or (iii) applying both (i) and (ii)	3410M	VGG-16 $\times$ 25	3450M
V100	100 variants of VGG-16 created as described above	13640M	VGG-16 $\times$ 100	13800M

Table 6.1: We experiment with ensembles of various sizes and network architectures.

Ensemble Networks. We experiment with ensembles of various convolutional architectures such as VGGNets, ResNets, Wide ResNets, and DenseNets. Ensembles of these architectures have been extensively used to evaluate fast ensemble training approaches (Lee et al. 2015a; Huang et al. 2017a). Each of these ensembles are composed of networks having diverse architectures as described in Table 6.1.

To provide a fair comparison with SE (where the snapshots have to be from the same network architecture), we create snapshots having comparable number of parameters to each of the ensembles described above. This comparable alternatives we used for SE are also summarized in Table 6.1.

For TN, we varied the number of shared layers and found that sharing the 3 initial layers provides the best accuracy. This is similar to the optimal proportion of shared layers in the TreeNets paper (Lee et al. 2015a). TN is not applicable to DenseNets or ResNets as it is designed only for networks without skip-connections (Lee et al. 2015a). We omit comparison with TN for such ensembles.

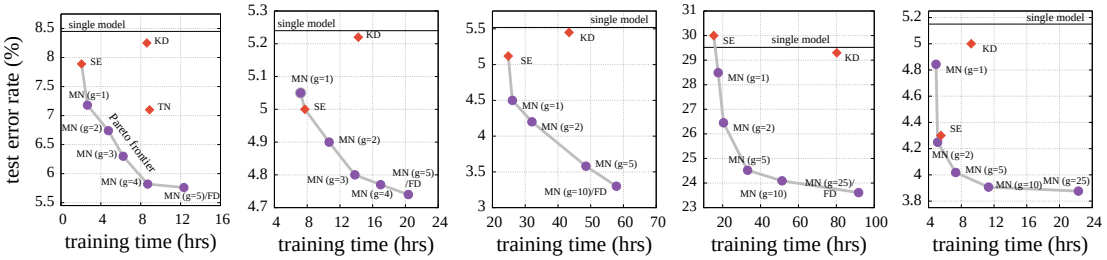
Training Setup. For all training approaches we use stochastic gradient descent with a mini-batch size of 256 and batch-normalization. All weights are initialized by sampling from a standard normal distribution. Training data is randomly shuffled before every training epoch. The learning rate is set to 0.1 with the exception of DenseNets. For DenseNets, we use a learning rate of 0.1 to train MotherNets and 0.01 to train hatched networks. This is inline with the learning rate decay used in the DenseNets paper (Huang et al. 2017b). For FD, KD, TN, and MotherNets, we stop training if the training accuracy does not improve for 15 epochs. For SE we use the optimized training setup proposed in the original paper (Huang et al. 2017a), starting with an initial learning rate of 0.2 and then training every snapshot for 60 epochs.

Data Sets. We experiment with a diverse array of data sets: SVHN, CIFAR-10, and CIFAR-100 (Krizhevsky 2009; Netzer et al.). The SVHN data set is composed of images of house numbers and has ten class labels. There are a total of 99K images. We use 73K for training and 26K for testing. The CIFAR-10 and CIFAR-100 data sets have 10 and 100 class labels respectively corresponding to various images of everyday objects. There are a total of 60K images – 50K training and 10K test images.

Hardware Platform. All experiments are run on the same server with Nvidia Tesla V100 GPU.

6.5.1 Better Accuracy-Training Time Tradeoff

We first show how MotherNets strike an overall superior accuracy-training time tradeoff when compared to existing fast ensemble training approaches. Figure 6.4 shows results across all our test data sets and ensemble networks. All graphs in Figure 6.4 depict the tradeoff between training time needed versus accuracy achieved. The core observation



(a) V5 (C-10) (b) D5 (C-10) (c) R10 (C-10) (d) V25 (C-100) (e) V25 (SVHN)

Figure 6.4: MotherNets provide consistently better accuracy-training time tradeoff when compared with existing fast ensemble training approaches across various data sets, architectures, and ensemble sizes.

from Figure 6.4 is that across all datasets and networks, MotherNets help establish a new Pareto frontier of this tradeoff. The different versions of MotherNets shown in Figure 6.4 represent different numbers of clusters used (g). When g=1, we use a single MotherNet, optimizing for training time, while when g becomes equal to the ensemble size, we optimize for accuracy (effectively this is equal to FD as every network is trained independently in its own cluster).

The horizontal line at the top of each graph indicates the accuracy of the best-performing single model in the ensemble trained from scratch. This serves as a benchmark and, in the vast majority of cases, all approaches do improve over a single model even when they have to sacrifice on accuracy to improve training time. MotherNets is consistently and significantly better than that benchmark.

Next we discuss each individual training approach and how it compares to MotherNets.

MotherNets vs. KD, TN, and BA. MotherNets (with g=1) is 2× to 4.2× faster than KD and results in up to 2 percent better test accuracy. KD suffers in terms of accuracy because its ensemble networks are more closely tied to the base network as they are trained from the output of the same network. KD's higher training cost is because distilling is expensive. Every network starts from scratch and is trained on the data set using a combination of empirical loss and the loss from the output of the teacher network. We observe that distilling a network still takes around 60 to 70 percent of the time required to train it using just the empirical loss.

	V5 C10	D5 C10	R10 C10	V25 C100	V25 SVHN
MN	96.71	97.43	98.61	87.5	97.17
SE	96.03	96.91	97.11	86.9	97.3

Table 6.2: MotherNets (with g=1) give better oracle test accuracy compared to Snapshot ensembles.

To achieve comparable accuracy to MotherNets (with g=1), TN requires up to 3.8× more training time on $V5$. In the same time budget, MotherNets can train with g=4 providing over one percent reduction in test error rate. The higher training time of TN is due to the fact that it combines several networks together to create a monolithic architecture with various branches. We observe that training this takes a significant time per epoch as well as requires more epochs to converge. Moreover, TN does not generalize to neural networks with skip-connections.

Figure 6.4 does not show results for BA because it is an outlier. BA takes on average 73 percent of the time FD needs to train but results in significantly higher test error rate than any of the baseline approaches including the single model. Compared to BA, MotherNets is on average 3.6 × faster and results in significantly better accuracy – up to 5.5 percent lower absolute test error rate. These observations are consistent with past studies that show how BA is ineffective when training deep neural networks as it reduces the number of unique data items seen by individual networks (Lee et al. 2015a).

Overall, the low test error rate of MotherNets when compared to KD, TN, and BA stems from the fact that transferring the learned function from MotherNets to target ensemble networks provides a good starting point as well as introduces regularization for further training. This also allows hatched ensemble networks to converge significantly faster, resulting in overall lower training time.

Training Time Breakdown. To better understand where the time goes during the training process, Figure 6.5 provides the time breakdown per ensemble network. We show this for the D5 ensemble and compare MotherNets (with g=1) with individual training approaches FD, BA, and KD. While other approaches spend significant time

training each network, MotherNets, can train these networks very quickly after having trained the core MotherNet (black part in the MotherNets stacked bar in Figure 6.5). We observe similar time breakdown across all ensembles in our experiments.

6.5.2 MotherNets vs. SE and Scaling to Large Ensembles

Across all experiments in Figure 6.4, SE is the closest baseline to MotherNets. In effect, SE is part of the very same Pareto frontier defined by MotherNets in the accuracy-training cost tradeoff. That is, it represents one more valid point that can be useful depending on the desired balance. For example, in Figure 6.4a (for V5 CIFAR-10), SE sacrifices nearly one percent in test error rate compared to MotherNets (with g=1) for a small improvement in training cost. We observe similar trends in Figure 6.4c and 6.4d). In Figure 6.4b, SE achieves a balance that is in between MotherNets with one and two clusters. However, when training V25 on SVHN (Figure 6.4e) SE is in fact outside the Pareto frontier as it is both slower and achieves worst accuracy.

Overall, MotherNets enables drastic improvements in either accuracy or training time compared to SE by being able to control and navigate the tradeoff between the two.

Oracle Accuracy. Also, Table 6.2 shows that MotherNets (with g=1) enable better oracle test accuracy when compared with SE across all our experiments. This is the accuracy if an *oracle* were to pick the prediction of the most accurate network in the ensemble per test element (Guzman-Rivera et al. 2012, 2014; Lee et al. 2015b). Oracle accuracy is an upper bound for the accuracy that any ensemble inference technique could achieve. This metric is also used to evaluate the utility of ensembles when they are applied to solve Multiple Choice Learning (MCL) problems (Guzman-Rivera et al. 2014; Lee et al. 2016; Brodie et al. 2018).

Scaling to Very Large Ensembles. As we discussed before, large ensembles help improve accuracy and thus ideally we would like to scale neural network ensembles to large number of models as it happens for other ensembles such as random forests (Oshiro et al. 2012; Bonab and Can 2016, 2017). Our previous results were for small to medium ensembles of 5, 10 or 25 networks. We now show that when it comes to larger ensembles, MotherNets dominate SE in both how accuracy and training time scale.

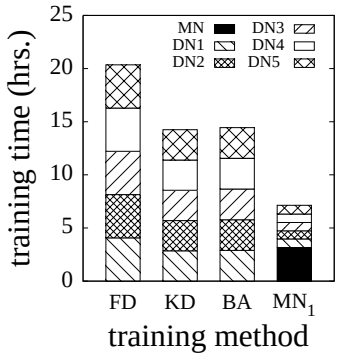


Figure 6.5: MotherNets train ensemble networks significantly faster after having trained the MotherNet (shown in black).

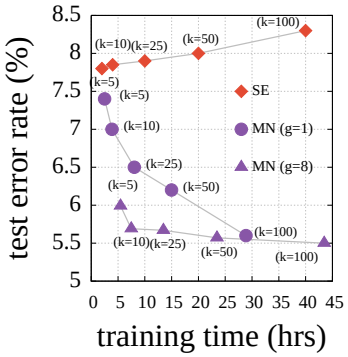


Figure 6.6: As ensemble size grows, MotherNets scale better than SE both in terms of training time and accuracy achieved.

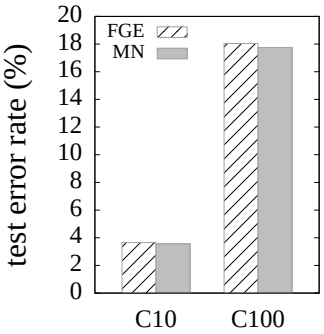


Figure 6.7: MotherNets outperform Fast Geometric Ensembles on Wide ResNet ensembles trained on CIFAR data sets.

Figure 6.6 shows results as we increase the number of networks up to a hundred variants of VGGNets trained on CIFAR-10. For every point in Figure 6.6, k indicates the number of networks. For MotherNets we plot results for the time-optimized version with g=1, as well as with g=8.

Figure 6.6 shows that as the size of the ensemble grows, MotherNets scale much better in terms of training time. Toward the end (for 100 networks), MotherNets train more than 10 hours faster (out of 40 total hours needed for SE). The training time of MotherNets grows at a much smaller rate because once the MotherNet has been trained, it takes 40 percent less time to train a hatched network than what it takes to train one snapshot.

In addition, Figure 6.6 shows that MotherNets does not only scale better in terms of training time, but also it scales better in terms of accuracy. As we add more networks to the ensemble, MotherNets keeps improving its error rate by nearly 2 percent while SE actually becomes worse by more than 0.5 percent. The declining accuracy of SE as the size of the ensemble increases has also been observed in the past, where by increasing the number of snapshots above six results in degradation in performance (i.e., test error rate) (Huang et al. 2017a).

Finaly, Figure 6.6 shows that different cluster settings for MotherNets allow us to achieve different performance balances while still providing robust and predictable navigation

of the tradeoff. In this case, with g=8 accuracy improves consistently across all points (compared to g=1) at the cost of extra training time.

6.5.3 Parallel Training

Deep learning pipelines rely on clusters of multiple GPUs to train computationally-intensive neural networks. MotherNets continue to improve training time in such cases when an ensemble is trained on more than one GPUs. We show this experimentally.

To train an ensemble of multiple networks, we queue all networks that are ready to be trained and assign them to available GPUs in the following fashion: If the number of ready networks is greater than free GPUs, then we assign a separate network to every GPU. If the number of ensemble networks available to be trained are less than the number of idle GPUs, then we assign one network to multiple GPUs dividing idle GPUs equally between networks. In such cases, we adopt data parallelism to train a network across multiple machines (Dean et al. 2012).

We train on a cluster of 8 Nvidia K80 GPUs and vary the number of available GPUs from 1 to 8. The training hyperparameters are the same as described in Section 3. Figure 6.8 and Figure 6.9 show the time to train the V5 and D5 ensembles respectively across FD, SE, and MotherNets. We observe that compared to Snapshot Ensembles, MotherNets (g=1) scale better as we increase the number of GPUs. The reason for this is that after the MotherNet has been trained, the rest of the ensemble networks are all ready to be trained. They can then be trained in a way that minimizes communication overhead by assigning them to as distinct set of GPUs as possible. Snapshot Ensembles, on the other hand, are generated one after the other. In a parallel setting this boils down to training a single network across multiple GPUs, which incurs communication overhead that increases as the number of GPUs increases (Keuper and Preundt 2016).

6.5.4 Improving Cloud Training Cost

One approach to speed up training of large ensembles is to utilize more than one machines. For example, we could train k individual networks in parallel using k machines.

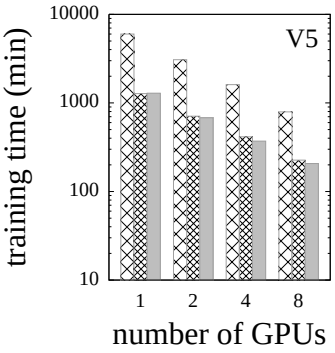


Figure 6.8: MotherNets continue to improve training cost in settings with multiple GPUs (V5).

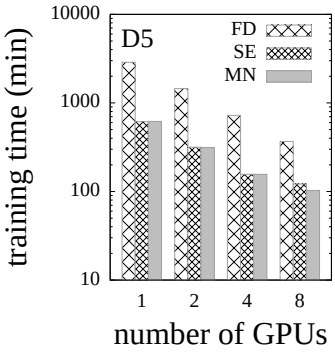


Figure 6.9: MotherNets is able to utilize multiple GPUs effectively scaling better than SE.

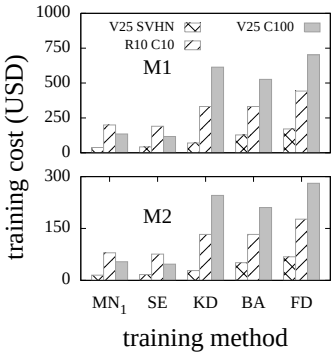


Figure 6.10: MotherNets result in dollar amount saving in cloud cost over other techniques.

While this does save time, the holistic cost in terms of energy and resources spent is still linear to the ensemble size.

One proxy for capturing the holistic cost is to look at the amount of money one has to pay on the cloud for training a given ensemble. In our next experiment, we compare all approaches using this proxy. Figure 6.10 shows the cost (in USD) of training on four cloud instances across two cloud service providers: (i) M1 that maps to AWS P2.xlarge and Azure NC6, and (ii) M2 that maps to AWS P3.2xlarge and Azure NCv3. M1 is priced at USD 0.9 per hour and M2 is priced at USD 3.06 per hour for both cloud service providers (Amazon 2019; Microsoft 2019).

6.5.5 Analyzing Ensemble Diversity

Next, we analyze how diverse are MotherNets ensembles compared to SE and FD.

Ensembles and Predictive Diversity. Theoretical results suggest that ensembles of models perform better when the models' predictions on a single example are less correlated. This is true under two assumptions: (i) models have equal correct classification probability and (ii) the ensemble uses majority vote for classification (Krogh and Vedelsby 1994; Rosen 1996; Kuncheva and Whitaker 2003). Under ensemble averag-

ing (the method we use to combine ensemble networks' predictions), no analytical proof that negative correlation reduces error rate exists, but lower correlation between models can be used to create a smaller upper bound on incorrect classification probability.

To establish this smaller upper bound, we can analyze how model covariance effects ensemble performance by using Chebyshev's Inequality to bound the chance that a model predicts an example incorrectly. By showing that lower covariance between models makes this bound on the probability smaller, we give an intuitive reason why ensembles with lower covariance between models perform better. The proof shows as well that the average model's predictive accuracy is important; finally, no assumptions need to be made for the proof to hold. The individual models can be of different quality and have different chances of getting each example correct.

Given a fixed training dataset, let Y_i be the softmax value of model i in the ensemble for the correct class, and let $\hat{Y} = \frac{1}{m}\sum_{i=1}^{m} Y_i$ be the ensemble's average softmax value on the correct class. Both are random variables with the randomness of $\hat{Y}$ and Y_i coming through the randomness of neural network training. Under the mild assumption that $E[\hat{Y}] > \frac{1}{2}$, so that the a one vs. all softmax classifier would say on average that the correct class is more likely, than Chebyshev's Inequality bounds the probability of incorrect prediction. Namely, the correct prediction is made with certainty if $\hat{Y} \geq \frac{1}{2}$ and so the probability of incorrect prediction is less than

$$P(|\hat{Y} - E[\hat{Y}]| \geq E[\hat{Y}] - \frac{1}{2}) \leq \frac{Var(\hat{Y})}{(E[\hat{Y}] - \frac{1}{2})^2}$$

From the form of the equation, we immediately see that keeping the average model accuracy $E[Y_i]$ high is important, and that degradation in model quality can offset reductions in variance. Since the variance of $\hat{Y}$ decomposes into $\frac{1}{m^2}(\sum_{i=1}^{m} Var(Y_i) + \sum_{i \neq i'} Cov(Y_i, Y_{i'}))$, we see that low model covariance keeps the variance of the ensemble low, and that models which have which have high covariance with other models provides little benefit to the ensemble.

Rapid Ensemble Training Methods. For MotherNets, as well as for all other compared techniques for ensemble training, the training procedure binds the models together

Full Data

0.025	0	0	0	0
0	0.025	0	0	0
0	0	0.027	0	0
0	0	0	0.026	−0.001
0	0	0	−0.001	0.026

MotherNets

0.027	0.01	0.009	0.009	0.009
0.01	0.027	0.009	0.009	0.008
0.009	0.009	0.028	0.009	0.008
0.009	0.009	0.009	0.028	0.008
0.009	0.008	0.008	0.008	0.028

Snapshot Ensemble

0.034	0.008	0.008	0.007	0.008
0.008	0.028	0.01	0.009	0.009
0.008	0.01	0.027	0.01	0.01
0.007	0.009	0.01	0.028	0.01
0.008	0.009	0.01	0.01	0.027

Figure 6.11: MotherNets (with g=1) train ensembles with lower model covariances compared to Snapshot Ensembles.

to decrease training time. This can have two negative effects compared to independent training of models:

1. by changing the model's architecture or training pattern, the technique affects each model's prediction quality (the model's marginal prediction accuracy suffers)
2. by sharing layers (TN), attempted softmax values (KD), or training epochs (SE, MN), the training technique creates positive correlations between model errors.

We compare here the magnitude of these two effects forMotherNets and Snapshot Ensembles when compared to independent training of each model on CIFAR-10 using V5.

Individual Model Quality. For both SE and MN, the individual model accuracy drops, but the effect is more pronounced in SE than MN. The mean misclassification percentage of the individual models for V5 using FD, MN and SE are 8.1%, 8.4% and 9.8% respectively. The poor performance of SE in this area is due to its difficulty in consistently hitting performant local minima, either because it overfits to the training data when trained for a long time or because its early snapshots need to be far away from the final optimum to encourage diversity.

Model Variance. Our goal in assessing variance is to see how the training procedure affects how models in the ensemble correlate with each other on each example. To do this, we train each of the five models in $V5$ five times under MN, SE, and FD.

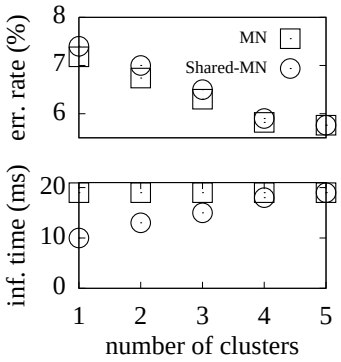


Figure 6.12: Shared MotherNets improve inference time by 2× for the V5 ensemble.

Letting Y_{ij} be the softmax of the correct model on test example j using model i, we then estimate $Var(Y_{ij})$ for each i, j and $Cov(Y_{ij}, Y_{i'j})$ for each i, i', j with $i \neq i'$ using the sample variance and covariance. To get a single number for a model, instead of one for each test example, we then average across all test examples, i.e. $Cov(Y_i, Y_{i'}) = \frac{1}{n}\sum_{j=1}^{n} Cov(Y_{ij}, Y_{i'j})$. For total variance numbers for the ensemble, we perform the same procedure on $Y_j = \frac{1}{5}\sum_{i=1}^{5} Y_{ij}$.

Figure 6.11 shows the results. As expected, independent training between the models in FD makes their corresponding covariance 0 and provides the greatest overall variance reduction for the ensemble, with ensemble variance at 0.0051. For both SE and MN, the covariance of separate models is non-zero at around 0.009 per pair of models; however, it is also significantly less than the variance of a single model. As a result, both MN and SE provide significant variance reduction compared to a single model. Whereas a single model has variance around 0.026, MN and SE provide ensemble variance of 0.0125 and 0.0130 respectively.

Takeaways. Since both SE and MN train nearly as fast as a single model, they provide variance reduction in prediction at very little training cost. Additionally, for MN, at the cost of higher training time, one can create more clusters and thus make the training of certain models independent of each other, zeroing out many of the covariance terms and reducing the overall ensemble variance. When compared to each other, MN with g=1 and SE have similar variance numbers, with MN slightly lower, but MotherNets

has a substantial increase in individual model accuracy when compared to Snapshot Ensembles. As a result, its overall ensemble performs better.

6.5.6 Enabling Fast Ensemble Inference

Figure 6.12 shows how shared-MotherNets improves inference time for an ensemble of 5 variants of VGGNet as described in Table 1. This ensemble is trained on the CIFAR-10 data set. We report both overall ensemble test error rate and the inference time per image. We see an improvement of 2× with negligible loss in accuracy. This improvement is because shared-MotherNets has a reduced number of parameters requiring less computation during inference time. This improvement scales with the ensemble size.

7

Conclusion and Future Work

This thesis presented Computation-Cautious Machine Learning Systems that improve various stages of the machine learning pipeline. The core intuition is to address the expensive bottleneck of repeated data movement and computation prevalent in machine learning pipelines. Computation-Cautious Machine Learning Systems identify opportunities to reuse, demystify, and share computation and data movement at various phases of machine learning pipelines. By doing so, they significantly speed up data exploration, improve model design under resource constraints, and establish a new Pareto frontier for the accuracy-training time tradeoff of deep neural network ensembles.

Future Directions

Expanding Scope of Tasks and Models. This thesis explores the application of Computation-Cautious Machine Learning Systems extensively to image classification using fully-connected and convolutional neural networks. There are opportunities to expand both the scope of tasks and models to apply Computation-Cautious Machine Learning Systems. For instance, we can expand Deep Collider to include (i) hetero-

geneous ensembles (e.g., various ratios of width-equivalent and depth-equivalent networks), (ii) arbitrary network architectures (including using network architecture search approaches), and (iii) application domains other than image classification such as object detection, machine translation, kernel methods, and deep generative models. Similarly, the paradigm of sharing computation and data movement between different models, as introduced in MotherNets, can be explored to train ensembles for object localization, machine translation, and time series prediction. Finally, MotherNets that enables fast training of a large set of heterogeneous networks can be explored in contexts other than ensemble learning that train multiple networks. These scenarios include network architecture search and hyperparameter tuning.

Feature Visualization. In addition to expanding the scope of tasks and models, we can extend the evaluation of MotherNets and Deep Collider to include visual analysis of features learned by different layers of the neural network models (Zeiler and Fergus 2014). In the case of MotherNets, this can provide a human-interpretable understanding of the diversity between the models. In the case of the Deep Collider, this can serve as yet another metric to understand the relative rank of various designs.

Understanding Model Design for Emerging Hardware. Recently, new hardware is being developed for deep learning. This hardware ranges from accelerators (such as Tensor Processing Units (TPU) by Google and Deep Learning Units (DLU) by Fujitsu) to hardware that uses quantum and photonics paradigms to improve computers' capabilities to move data and perform computation on it. For instance, photonics-electronics hybrid computers promise to provide orders of magnitude higher bandwidth between memory and compute than existing GPUs. Computation-Cautious Machine Learning Systems study and model various tradeoffs between compute and data movement and there are opportunities to extendx them to emerging hardware. There are important challenges of figuring out how to design, train, and deploy models (that are currently intended for existing hardware) to utilize these new capabilities properly: (i) How does the relationship between the size of a deep learning model – width, depth, and the number of parameters – and the resources it requires changes with new hardware? (ii) How to effectively distribute model training between different compute nodes? And (iii) How to set training hyperparameters such as batch size, weight decay, and learning rate?

Enabling Machine Learning for Low Resource Scenarios. There are challenges of efficiently applying machine learning to low-resource scenarios, e.g., in the global south with drastically less compute and memory resources available per capita and unreliable power and internet connectivity. One direction would be to design systems that coalesce various operators in a deep learning model to reduce both data access and computation during training and inference. Another direction is to design systems that can train and deploy deep learning models in unpredictable environments with frequent power cuts and variable compute resource availability. Finally, there are opportunities to conduct large-scale research in characterizing the diversity in training and deployment devices around the globe and the implications this has for model and hardware design. Deep Collider, for instance, can help study the holistic design space between models and hardware design, specifically in low-resource scenarios.

Incorporating Responsibility in Machine Learning Pipelines. When we apply machine learning pipelines to user-facing applications, then there are new challenges to consider. For example, when we use deep learning to make decisions that can directly impact humans, it is crucial to understand how deep learning models made these decisions and why. There are several opportunities for Deep Collider to understand better the relationship between various model designs and robustness and interpretability (Wasay et al. 2021). We can expand Deep Collider to include robustness, fairness, and interpretability metrics. In addition to this, there are opportunities for designing systems that can efficiently track the end-to-end provenance of machine learning pipelines making it easier to verify the degree of fairness they exhibit. Here, Computation-Cautious Machine Learning Systems' framework to efficiently use data movement and computation is ever more crucial to maintain interactivity as users debug and understand deployed pipelines.

Overall, this thesis paves the way for thinking holistically about machine learning pipelines, where we bring the resources required to run these pipelines – time, memory usage, cloud cost – to the forefront, along with the quality of the model. As machine learning pipelines scale to more data, to more applications, and ultimately to more people, a holistic approach to these pipelines enables more efficient utilization of resources while achieving desired outcomes.

Bibliography

NumPy. http://www.numpy.org, 2013.

Psycopg. http://initd.org/psycopg/, 2014.

National Centers for Environmental Information (NCEI). https://www.ncei.noaa.gov, 2016.

Wofram – Descriptive Statistics. https://reference.wolfram.com/language/tutorial/DescriptiveStatistics.html, 2017.

Azza Abouzied, Joseph M Hellerstein, and Avi Silberschatz. Playful Query Specification with DataPlay. *Proceedings of the VLDB Endowment*, 5(12):1938–1941, 2012. URL http://vldb.org/pvldb/vol5/p1938{_}azzaabouzied{_}vldb2012.pdf.

Jan Adler. *Materialized views in distributed key-value stores*. PhD thesis, Technical University of Munich, Germany, 2020. URL https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20200714-1546769-1-3.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013. ISBN 9781450319942. doi: 10.1145/2465351.2465355. URL http://dl.acm.org/citation.cfm?id=2465351.2465355.

Dimitris K Agrafiotis, Walter Cedeno, and Victor S Lobanov. On the use of neural network ensembles in qsar and qspr. *Journal of chemical information and computer sciences*, 42(4), 2002.

Rafi Ahmed, Randall G. Bello, Andrew Witkowski, and Praveen Kumar. Automated generation of materialized views in oracle. *Proc. VLDB Endow.*, 13(12):3046–3058, 2020. doi: 10.14778/3415478.3415533. URL http://www.vldb.org/pvldb/vol13/p3046-ahmed.pdf.

Foteini Alvanaki and Sebastian Michel. Tracking set correlations at large scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1507–1518, 2014. doi: 10.1145/2588555.2610510. URL http://doi.acm.org/10.1145/2588555.2610510.

Amazon. Aws pricing. https://aws.amazon.com/pricing/, 2019. (Accessed on 05/16/2019).

Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *CoRR*, abs/1802.06509, 2018.

Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, 2014.

Daniel Barbara and Mark Sullivan. Quasi-cubes: Exploiting Approximations in Multidimensional Databases. *ACM SIGMOD Record*, 26(3):12–17, 1997. doi: 10.1145/262762.262764. URL http://doi.acm.org/10.1145/262762.262764.

Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1363–1375, 2016. doi: 10.1145/2882903.2882919. URL http://doi.acm.org/10.1145/2882903.2882919.

Yoshua Bengio, Ian J Goodfellow, and Aaron Courville. Deep learning. *Nature*, 521, 2015.

Kevin Beyer and Raghu Ramakrishnan. Bottom-up Computation of Sparse and Iceberg CUBE. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999. ISBN 1-58113-084-8. doi: 10.1145/304182.304214. URL http://doi.acm.org/10.1145/304182.304214.

Christopher M Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

Matthias Boehm, Arun Kumar, and Jun Yang. Data management in machine learning systems. *Synthesis Lectures on Data Management*, 11(1):1–173, 2019.

Hamed R Bonab and Fazli Can. A theoretical framework on the ideal number of classifiers for online ensembles in data streams. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016.

Hamed R Bonab and Fazli Can. Less is more: A comprehensive framework for the number of components of ensemble classifiers. *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

Peter Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, 1999. URL http://www.vldb.org/conf/1999/P5.pdf.

Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

M. Brodie, C. Tensmeyer, W. Ackerman, and T. Martinez. Alpha model domination in multiple choice learning. In *IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018.

Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher R. Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 194–205, 2016. doi: 10.1145/2854038.2854042. URL http://doi.acm.org/10.1145/2854038.2854042.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI Conference on Artificial Intelligence*, 2018.

Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014. doi: 10.14778/2735496.2735503.

Sirish Chandrasekaran, Mehul A. Shah, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sammuel R. Madden, and Fred Reiss. TelegraphCQ: continuous dataflow processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003. URL http://dl.acm.org/citation.cfm?id=872757.872857.

Surajit Chaudhuri and Vivek R Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998. doi: 10.1145/276304.276337. URL http://doi.acm.org/10.1145/276304.276337.

Surajit Chaudhuri, Rajeev Motwani, and Vivek R Narasayya. Random Sampling for Histogram Construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 436–447, 1998. doi: 10.1145/276304.276343. URL http://doi.acm.org/10.1145/276304.276343.

Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. Effective Use of Block-Level Sampling in Statistics Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 287–298, 2004. doi: 10.1145/1007568.1007602. URL http://doi.acm.org/10.1145/1007568.1007602.

Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. In *International Conference on Learning Representations (ICLR), San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Y Chen, A Rau-Chaplin, F Dehne, T Eavis, D Green, and E Sithirasenan. cgmOLAP: Efficient Parallel Generation and Querying of Terabyte Size ROLAP Data Cubes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 164, 2006. doi: 10.1109/ICDE.2006.32.

Nadezhda Chirkova, Ekaterina Lobacheva, and Dmitry Vetrov. Deep ensembles on a fixed memory budget: One wide network or several thinner ones?, 2020.

Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2011.

Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

Christopher Culley, Supreeta Vijayakumar, Guido Zampieri, and Claudio Angione. A mechanism-aware and multiomic machine-learning pipeline characterizes yeast cell growth. *Proceedings of the National Academy of Sciences*, 117(31):18869–18879, 2020.

Nilesh N Dalvi and Dan Suciu. Answering Queries from Statistics and Probabilistic Views. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 805–816, 2005. URL http://www.vldb2005.org/program/paper/thu/p805-dalvi.pdf.

Yann N Dauphin and Yoshua Bengio. Big neural networks waste capacity. *International Conference on Learning Representations (ICLR)*, 2013.

Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

Christopher De Sa and Matthew Feldman. Understanding and optimizing asynchronous low-precision stochastic gradient descent.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, 2012.

Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F Naughton. Caching Multidimensional Queries Using Chunks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1998. doi: 10.1145/276304.276328. URL http://doi.acm.org/10.1145/276304.276328.

Thomas G Dietterich. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*, 2000.

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-Example: An Automatic Query Steering Framework for Interactive Data Exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2014. doi: 10.1145/2588555.2610523. URL http://doi.acm.org/10.1145/2588555.2610523.

Pedro Domingos. Why does bagging work? a bayesian account and its implications. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, 1999.

Chao Dong, Yubin Deng, Chen Change Loy, and Xiaoou Tang. Compression artifacts reduction by a deep convolutional network. In *IEEE International Conference on Computer Vision*, 2015.

Marina Drosou and Evaggelia Pitoura. YmalDB: A Result-Driven Recommendation System for Databases. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 725–728, 2013. doi: 10.1145/2452376.2452464. URL http://doi.acm.org/10.1145/2452376.2452464.

Harris Drucker, Robert Schapire, and Patrice Simard. Improving performance in neural networks using a boosting algorithm. In *Advances in Neural Information Processing Systems*, 1993.

Li Du, Yuan Du, Yilei Li, Junjie Su, Yen-Cheng Kuan, Chun-Chen Liu, and Mau-Chung Frank Chang. A reconfigurable streaming deep convolutional neural network accelerator for internet of things. *IEEE Transactions on Circuits and Systems*, 2017.

Curtis E Dyreson. Information Retrieval from an Incomplete Data Cube. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 532–543, 1996. ISBN 1-55860-382-4. URL http://dl.acm.org/citation.cfm?id=645922.673326.

David Eigen, Jason Rolfe, Rob Fergus, and Yann LeCun. Understanding deep architectures using a recursive convolutional network. *International Conference on Learning Representations (ICLR)*, 2013.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. *CoRR*, abs/1512.03965, 2015.

Ying Feng, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. Range cube: efficient cube computation by exploiting data correlation. In *Proceedings. 20th International Conference on Data Engineering*, pages 658–669, 2004. ISBN 0-7695-2065-0. doi: 10.1109/ICDE.2004.1320035. URL http://ieeexplore.ieee.org/document/1320035/.

Sloan Foundation. Sloan digital sky survey. URL http://www.sdss.org.

The R Foundation. The R Project for Statistical Computing. https://www.r-project.org, 2016.

Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P. Vetrov, and Andrew Gordon Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*, 2018.

Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1): 57–72, 2007. doi: 10.1109/TKDE.2007.250585. URL http://dx.doi.org/10.1109/TKDE.2007.250585.

Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 466–475, 1997. URL http://www.vldb.org/conf/1997/P466.PDF.

Phillip B Gibbons, Viswanath Poosala, Swarup Acharya, Yair Bartal, Yossi Matias, S Muthukrishnan, Sridhar Ramaswamy, and Torsten Suel. AQUA: System and Techniques for Approximate Query Answering. *Bell Labs - Technical Report*, 1998.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Anjana Gosain and Kavita Sachdeva. Selection of materialized views using stochastic ranking based backtracking search optimization algorithm. *International journal of system assurance engineering and management*, 10(4):801–810, 2019.

Pablo M Granitto, Pablo F Verdes, and H Alejandro Ceccatto. Neural network ensembles: Evaluation of aggregation algorithms. *Artificial Intelligence*, 163(2), 2005.

Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining*

and Knowledge Discovery, 1(1):29–53, 1997. doi: 10.1023/A:1009726021843. URL http://dx.doi.org/10.1023/A:1009726021843.

Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, 1995. doi: 10.1145/223784.223849. URL http://doi.acm.org/10.1145/223784.223849.

Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, 316(22), 2016.

Philip Jia Guo. *Software Tools to Facilitate Research Programming*. PhD thesis, Stanford University, 2012. URL https://searchworks.stanford.edu/view/9625286.

Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *IEEE International Conference on Data Mining (ICDM)*, 2016.

Abner Guzman-Rivera, Dhruv Batra, and Pushmeet Kohli. Multiple choice learning: Learning to produce multiple structured outputs. In *Advances in Neural Information Processing Systems*, 2012.

Abner Guzman-Rivera, Pushmeet Kohli, Dhruv Batra, and Rob Rutenbar. Efficiently enforcing diversity in multi-output structured prediction. In *Artificial Intelligence and Statistics*, 2014.

Alon Y Halevy. Structures, Semantics and Statistics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 4–6, 2004. URL http://www.vldb.org/conf/2004/KEY2.PDF.

Pat Hanrahan. VizQL: A Language for Query, Analysis and Visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 721, 2006. doi: 10.1145/1142473.1142560. URL http://doi.acm.org/10.1145/1142473.1142560.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Joseph M Hellerstein and Jeffrey F Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 423–434, 1996. doi: 10.1145/233269.233359. URL http://doi.acm.org/10.1145/233269.233359.

Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

Bill Howe, Francois Ribalet, Daniel Halperin, Sagar Chitnis, and E Virginia Armbrust. Sqlshare: Scientific workflow via relational view sharing.

Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2013. doi: 10.1145/2463676.2465273. URL http://doi.acm.org/10.1145/2463676.2465273.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision*, 2016.

Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger. Snapshot ensembles: Train 1, get m for free. *5th International Conference on Learning Representations (ICLR)*, 2017a.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition, (CVPR)*, 2017b.

Thomas Huang. Computer vision: Evolution and promise. 1996.

Jonathan Huggins, Trevor Campbell, and Tamara Broderick. Coresets for scalable bayesian logistic regression. In *Advances in Neural Information Processing Systems*, 2016.

Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: Near-linear acceleration of deep neural network training on compute clusters. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Stratos Idreos. *Big Data Exploration*. Taylor and Francis, 2013.

Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, pages 277–281, 2015. URL http://dl.acm.org/citation.cfm?id=2723372.2731084.

Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *IEEE Annual International Symposium on Computer Architecture*, 2018.

Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–293, 2016. doi: 10.1145/2882903.2882957. URL http://doi.acm.org/10.1145/2882903.2882957.

Cheng Ju, Aurélien Bibaut, and Mark J van der Laan. The relative performance of ensemble methods with deep convolutional neural networks for image classification. *CoRR*, abs/1704.01664, 2017.

Minsuk Kahng, Dezhi Fang, and Duen Horng (Polo) Chau. Visual Exploration of Machine Learning Results Using Data Cube Analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*, pages 1:1—-1:6, 2016. ISBN 978-1-4503-4207-0. doi: 10.1145/2939502.2939503. URL http://doi.acm.org/10.1145/2939502.2939503.

Yashal Shakti Kanungo, Bhargav Srinivasan, and Savita Choudhary. Detecting diabetic retinopathy using deep learning. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 801–804. IEEE, 2017.

Arthur M Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal*, 5(1):35–47, 1996. doi: 10.1007/s007780050014. URL http://dx.doi.org/10.1007/s007780050014.

Janis Keuper and Franz-Josef Preundt. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 19–26. IEEE, 2016.

Alicia Key, Bill Howe, Daniel Perry, and Cecilia R Aragon. VizDeck: self-organizing dashboards for visual analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 681–684, 2012. doi: 10.1145/2213836.2213931. URL http://doi.acm.org/10.1145/2213836.2213931.

Yoon Kim. Convolutional neural networks for sentence classification. *Empirical Methods in Natural Language Processing*, 2014.

Dan Kondratyuk, Mingxing Tan, Matthew Brown, and Boqing Gong. When ensembling smaller models is more efficient than single large models, 2020.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation and active learning. In *International Conference on Neural Information Processing Systems*, 1994.

Amit Kumar and TV Vijay Kumar. Materialized view selection using set based particle swarm optimization. *International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, 12(3):18–39, 2018.

Ludmila I. Kuncheva and Christopher J. Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Machine Learning*, 51 (2), 2003.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553), 2015.

Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M heads are better than one: Training a diverse ensemble of deep networks. *CoRR*, abs/1511.06314, 2015a.

Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M heads are better than one: Training a diverse ensemble of deep networks. *CoRR*, abs/1511.06314, 2015b.

Stefan Lee, Senthil Purushwalkam Shiva Prakash, Michael Cogswell, Viresh Ranjan, David Crandall, and Dhruv Batra. Stochastic multiple choice learning for training diverse deep ensembles. In *Advances in Neural Information Processing Systems*, 2016.

Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. 1966.

Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F Naughton, and Jignesh M Patel. Tuple-oriented compression for large-scale mini-batch stochastic gradient descent. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1517–1534, 2019a.

Side Li, Lingjiao Chen, and Arun Kumar. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1571–1588, 2019b.

Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. Sampling Cube: A Framework for Statistical OLAP over Sampling Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 779–790, 2008. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376695. URL http://doi.acm.org/10.1145/1376616.1376695.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

Erietta Liarou and Stratos Idreos. dbTouch in action database kernels for touch-based data exploration. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1262–1265, 2014. doi: 10.1109/ICDE.2014.6816756. URL http://dx.doi.org/10.1109/ICDE.2014.6816756.

Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin Kersten. Enhanced stream processing in a DBMS kernel. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 501–512, 2013. ISBN

9781450315975. doi: 10.1145/2452376.2452435. URL http://dl.acm.org/citation.cfm?id=2452376.2452435.

Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International journal of computer vision*, 128(2):261–318, 2020.

Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3):421–430, 2013. doi: 10.1111/cgf.12129. URL http://dx.doi.org/10.1111/cgf.12129.

Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*. 2017.

J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

David Madigan and Ron Wasserstein. Statistics and science. *London Workshop on the Future of the Statistical Sciences*, 2013.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: when is deep better than shallow. *CoRR*, abs/1603.00988, 2016.

Microsoft. Pricing - windows virtual machines — microsoft azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/, 2019. (Accessed on 05/16/2019).

Tom M. Mitchell, William W. Cohen, Estevam R. Hruschka Jr., Partha P. Talukdar, Bo Yang, Justin Betteridge, Andrew Carlson, Bhavana Dalvi Mishra, Matt Gardner, Bryan Kisiel, Jayant Krishnamurthy, Ni Lao, Kathryn Mazaitis, Thahir Mohamed, Ndapandula Nakashole, Emmanouil A. Platanios, Alan Ritter, Mehdi Samadi, Burr Settles, Richard C. Wang, Derry Wijaya, Abhinav Gupta, Xinlei Chen, Abulhair Saparov, Malcolm Greaves, and Joel Welling. Never-ending learning. volume 61, 2018.

Erick Moen, Dylan Bannon, Takamasa Kudo, William Graf, Markus Covert, and David Van Valen. Deep learning for cellular image analysis. *Nature methods*, 16(12):1233–1246, 2019.

Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998. URL http://dl.acm.org/citation.cfm?id=645924.671173.

Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, 2010. doi: 10.1145/1807167.1807188. URL http://doi.acm.org/10.1145/1807167.1807188.

Hannes Mühleisen and Thomas Lumley. Best of Both Worlds: Relational Databases and Statistics. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 32:1—-32:4, 2013. doi: 10.1145/2484838.2484869. URL http://doi.acm.org/10.1145/2484838.2484869.

Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, 1997. ISBN 0-89791-911-4. doi: 10.1145/253260.253277. URL http://doi.acm.org/10.1145/253260.253277.

Arnab Nandi. Querying Without Keyboards. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013. URL http://www.cidrdb.org/cidr2013/Papers/CIDR13{_}Paper37.pdf.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning.

Thong Q Nguyen, Daniel Weitekamp, Dustin Anderson, Roberto Castello, Olmo Cerri, Maurizio Pierini, Maria Spiropulu, and Jean-Roch Vlimant. Topology classification with deep learning to improve real-time event selection at the lhc. *Computing and Software for Big Science*, 3(1):1–14, 2019.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, 2016.

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *CoRR*, abs/1106.5730, 2011. URL http://arxiv.org/abs/1106.5730.

Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. *International Conference on Learning Representations (ICLR)*, 2018.

Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. How many trees in a random forest? In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2012.

Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. pages 2–4, 2005.

Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 651–665, 2016. doi: 10.1145/2872362.2872415. URL http://doi.acm.org/10.1145/2872362.2872415.

Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

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

Bruce E. Rosen. Ensemble learning using decorrelated neural networks. *Connection Science*, 1996.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115 (3), 2015.

Sunita Sarawagi and Gayatri Sathe. I3: Intelligent, Interactive Investigation of OLAP Data Cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 589—-, 2000. ISBN 1-58113-217-4. doi: 10.1145/342009.336564. URL http://doi.acm.org/10.1145/342009.336564.

Saket Sathe and Karl Aberer. AFFINITY: Efficiently Querying Statistical Measures on Time-Series Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 841–852, 2013. doi: 10.1109/ICDE.2013.6544879. URL http://dx.doi.org/10.1109/ICDE.2013.6544879.

James B. Saxe and Jon Louis Bentley. Transforming Static Data Structures to Dynamic Structures. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 148–168, 1979.

Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2882939. URL http://doi.acm.org/10.1145/2882903.2882939.

SciDB. SciDB-Py. *http://scidb-py.readthedocs.io/en/stable/*, 2016.

Christopher J Shallue and Andrew Vanderburg. Identifying exoplanets with deep learning: A five-planet resonant chain around kepler-80 and an eighth planet around kepler-90. *The Astronomical Journal*, 155(2):94, 2018.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations (ICLR)*, 2017.

Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep searning in medical image analysis. *Annual review of biomedical engineering*, 19, 2017.

Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering Queries Based on Example Tuples. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 2014. doi: 10.1145/2588555.2593664. URL http://doi.acm.org/10.1145/2588555.2593664.

Lefteris Sidirourgos, Martin L Kersten, and Peter A Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11{_}Paper39.pdf.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

Saurabh Singh, Derek Hoiem, and David Forsyth. Swapout: Learning an ensemble of deep architectures. In *Advances in Neural Information Processing Systems*, 2016.

Mohammad Karim Sohrabi and Hossein Azgomi. Evolutionary game theory approach to materialized view selection in data warehouses. *Knowledge-Based Systems*, 163: 558–571, 2019.

Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *IEEE International Conference on Data Engineering (ICDE)*, 2017.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 2014.

Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(1):52–65, 2002. doi: 10.1109/2945.981851. URL http://doi.ieeecomputersociety.org/10.1109/2945.981851.

Michael Stonebraker and Joseph Kalash. TIMBER: A Sophisticated Relation Browser (Invited Paper). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10, 1982. URL http://www.vldb.org/conf/1982/P001.PDF.

Chaudhuri Surajit. Data Exploration Challenges in the Age of Big Data. In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*, 2016. URL http://db.cs.pitt.edu/birte2016/keynote.html.

Vivienne Sze, Yu-Hsin Chen, Joel S. Einer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. In *IEEE Custom Integrated Circuits Conference (CICC)*, 2017a.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12): 2295–2329, 2017b.

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

Matus Telgarsky. Benefits of depth in neural networks. *Conference on Learning Theory (COLT*, 2016.

Gregor Urban, Krzysztof J Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. Do deep convolutional nets really need to be deep and convolutional? *International Conference on Learning Representations (ICLR)*, 2016.

Mark J Van der Laan, Eric C Polley, and Alan E Hubbard. Super learner. *Statistical applications in genetics and molecular biology*, 6(1), 2007.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, 2013.

Yu Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. 2020.

Abdul Wasay and Stratos Idreos. More or less: When and how to build neural network ensembles. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=z5Z023VBmDZ. under review.

Abdul Wasay, Manos Athanassoulis, and Stratos Idreos. Queriosity: Automated Data Exploration. In *Proceedings of the IEEE International Congress on Big Data*, pages 716–719, 2015. doi: 10.1109/BigDataCongress.2015.116. URL http://dx.doi.org/10.1109/BigDataCongress.2015.116.

Abdul Wasay, Manos Athanassoulis, and Stratos Idreos. Queriosity: Automated data exploration. In *Proceedings of IEEE International Congress on Big Data*, 2015.

Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2017.

Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

Abdul Wasay, Subarna Chatterjee, and Stratos Idreos. Deep learning: Systems and responsibility. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2021.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International Conference on Machine Learning*, 2016.

Tao Wei, Changhu Wang, and Chang Wen Chen. Modularized morphing of neural networks. *CoRR*, abs/1701.03281, 2017.

Charles Weill, Javier Gonzalvo, Vitaly Kuznetsov, Scott Yang, Scott Yak, Hanna Mazzawi, Eugen Hotaj, Ghassen Jerfel, Vladimir Macko, Ben Adlam, Mehryar Mohri, and Corinna Cortes. Adanet: A scalable and flexible framework for automatically learning ensembles. *CoRR*, abs/1905.00080, 2019.

Bernard Widrow et al. *Adaptive" adaline" Neuron Using Chemical" memistors.".* 1960.

M. L. Williams, K. M. Fischer, J. T. Freymueller, B. Tipoff, and A. M. TrÃ©hu. An earthscope science plan 2010-2020, feb 2010. URL http://www.earthscope.org/assets/uploads/pages/es_sci_plan.pdf.

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

Eugene Wu, Leilani Battle, and Samuel R. Madden. The case for data visualization management systems. *Proceedings of the VLDB Endowment*, 7(10):903–906, 2014. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2732951.2732964.

Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. Continuous Sampling for Online Aggregation Over Multiple Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 651–662, 2010. doi: 10.1145/1807167.1807238. URL http://doi.acm.org/10.1145/1807167.1807238.

Tianyi Wu, Dong Xin, and Jiawei Han. ARCube: Supporting Ranking Aggregate Queries in Partially Materialized Data Cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–92, 2008. doi: 10.1145/1376616.1376627. URL http://doi.acm.org/10.1145/1376616.1376627.

X Xie, X Hao, T B Pedersen, P Jin, and J Chen. OLAP Over Probabilistic Data Cubes I: Aggregating, Materializing, and Querying. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 799–810, 2016. doi: 10.1109/ICDE.2016.7498291.

Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W Wah. Star-cubing: Computing Iceberg Cubes by Top-down and Bottom-up Integration. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 2003. ISBN 0-12-722442-4. URL http://dl.acm.org/citation.cfm?id=1315451.1315493.

Li Xu, Jimmy SJ Ren, Ce Liu, and Jiaya Jia. Deep convolutional neural network for image deconvolution. In *Advances in Neural Information Processing Systems*, 2014.

Meng Ye and Yuhong Guo. Self-training ensemble networks for zero-shot image recognition. *Knowl.-Based Syst.*, 123:41–60, 2017.

Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword Search in Relational Databases: A Survey. *IEEE Data Engineering Bulletin*, 33(1):67–78, 2010. URL http://sites.computer.org/debull/A10mar/yu-paper.pdf.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *Proceedings of the British Machine Vision Conference*, 2016.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

Shengjia Zhao, Hongyu Ren, Arianna Yuan, Jiaming Song, Noah Goodman, and Stefano Ermon. Bias and generalization in deep generative models: An empirical study. In *Advances in Neural Information Processing Systems*, 2018.

Yihong Zhao, Prasad Deshpande, and Jeffrey F Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170, 1997. doi: 10.1145/253260.253288. URL http://doi.acm.org/10.1145/253260.253288.

Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *International workshop on Peer-To-Peer Systems, IPTPS 2006, Santa Barbara, CA, USA, February 27-28, 2006*.

Yunyue Zhu and Dennis Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 358–369, 2002. URL http://www.vldb.org/conf/2002/S10P04.pdf.

Jia Zou. Using deep learning models to replace large materialized views in relational database. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL http://cidrdb.org/cidr2021/papers/cidr2021_abstract05.pdf.

List of Figures