Materialization Trade-offs for Feature Transfer from Deep CNNs for Multimodal Data Analytics

Supun Nakandala Arun Kumar
University of California, San Diego
{snakanda, arunkk}@eng.ucsd.edu

ABSTRACT

Deep convolutional neural networks (CNNs) achieve near-human accuracy on many image understanding tasks. This has led to a growing interest in using deep CNNs to integrate images with structured data for *multimodal analytics* in many applications to improve prediction accuracy. Since training deep CNNs from scratch is expensive and laborious, *transfer learning* has become popular: using a pre-trained CNN, one reads off a certain layer of features to represent images and combines them with other features for a downstream ML task. Since no single layer will always offer best accuracy in general, such *feature transfer* requires comparing many CNN layers. The current dominant approach to this process on top of scalable analytics systems such as TensorFlow and Spark is fraught with inefficiency due to redundant CNN inference and the potential for system crashes due to manual memory management. We present Vista, the first data system to mitigate such issues by elevating the feature transfer workload to a declarative level and formalizing the data model of CNN inference. Vista enables automated optimization of *feature materialization trade-offs*, memory usage, and system configuration. Experiments with real-world datasets and deep CNNs show that apart from enabling seamless feature transfer, Vista helps avoid system crashes and also reduces runtimes by 67%–90%.

1. INTRODUCTION

Deep convolutional neural networks (CNNs) have revolutionized computer vision, yielding near-human accuracy for many image understanding tasks [52]. The key technical reason for their success is how they extract a hierarchy of relevant parametrized features from images, with the parameters learned automatically during training [37]. Each layer of features captures a different level of abstraction about the image, e.g., low-level edges and patterns in the lowest layers to abstract object shapes in the highest layers. This remarkable ability of deep CNNs is illustrated in Figure 1(A).

The success of deep CNNs presents an exciting opportunity to holistically integrate image data into traditional data analytics applications in the enterprise, Web, healthcare, and other domains that have hitherto relied mainly on structured data features but had auxiliary images that were not exploited. For instance, product recommendation systems such as Amazon are powered by ML algorithms that relied mainly on structured data features such as price, vendor, purchase history, etc. Such applications are increasingly using deep CNNs to exploit product images by extracting visually-relevant features to help improve ML accuracy, especially for products such as clothing and footwear [55]. Indeed, such deep CNN-based feature extraction already powers visual search and analytics at some Web companies [42]. Numerous other applications could also benefit from such "multimodal" analytics, including inventory management, healthcare, and online advertising [21].

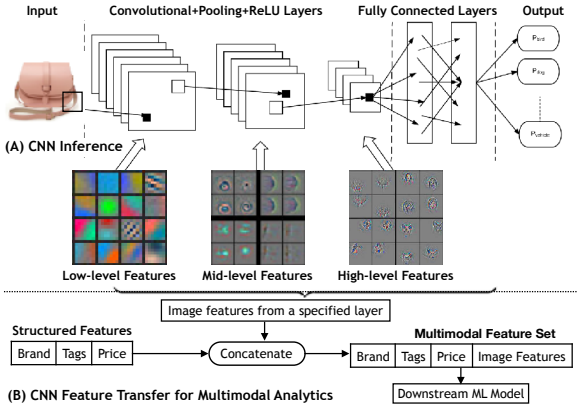


Figure 1: (A) Simplified illustration of a typical deep CNN and its hierarchy of learned features (based on [75]). (B) Illustration of CNN feature transfer for multimodal analytics.

Since training deep CNNs from scratch is expensive in terms of resource costs (e.g., one might need many GPUs [3]) and the number of labeled examples needed, an increasingly popular paradigm for handling images is *transfer learning* [59]. Essentially, one uses a pre-trained deep CNN, e.g., ImageNet-trained AlexNet [34, 47] and reads off a certain layer of the features it produces on an image as the image's representation [18, 35]. Any downstream ML model can use these image features along with the structured features, say, the popular logistic regression model or even "shallow" neural networks. Figure 1(B) illustrates this process. Thus, such *feature transfer* helps reduce costs dramatically for using deep CNNs. Indeed, this paradigm is responsible for many high-profile successes of CNNs, including for detecting cancer [36] and diabetic retinopathy [68], facial analyses [20], and for product recommendations and search [42, 56].

Alas, feature transfer creates a new bottleneck for data scientists in practice: it is impossible to say in general which layer of a CNN will yield the best accuracy for the downstream ML task [27]. The common guideline is to extract and compare multiple layers of CNN features [27, 72]. This is a *model selection* process that combines CNN features and structured data [48]. Perhaps surprisingly, the current dom-

inant approach to handling feature transfer at scale is for data scientists to manually *materialize* each CNN layer from scratch as flat files using tools such as TensorFlow [24], load such data into a scalable data analytics system for downstream ML tasks, say, using Spark and MLlib [57], which is increasingly popular among enterprises [4]. Apart from reducing the productivity of data scientists, such manual management of feature transfer workloads leads to wasted opportunities to reuse and optimize computations, which raises runtimes and in turn, costs, especially in the cloud.

In this work, we aim to resolve the above issues for large-scale feature transfer with deep CNNs for multimodal data analytics. We start with a simple but crucial observation: the different layers of a typical CNN are not independent– *extracting a higher layer requires a superset of the computations needed for a lower layer.* Thus, instead of materializing all layers from scratch, we can reuse previously created layers, subject to other system constraints such as memory or storage. This is a novel instance of a classical database systems-style concern: *materialization trade-offs.*

At first blush, feature transfer might seem straightforward: *Why not materialize and cache all layers of interest in one go and use a layer as needed?* While this reduces runtimes, as observed above, it increases *memory pressure*, since CNN features are often orders of magnitude larger than the input (e.g., one of ResNet50's layers is 784kB, while the input is 14kB [38]). Such data blowup lead to non-trivial systems trade-offs for handling memory usage at scale. In fact, performed naively, it could cause system crashes, which would frustrate data scientists and raise costs by forcing them to manually tweak the system or use needlessly more expensive machines. Also, caching unused layers can cause needless *disk spills*, which raises runtimes further. Thus, overall, large-scale feature transfer is technically challenging due to two key systems-oriented concerns: *reliability* (avoiding system crashes) and *efficiency* (reducing runtimes).

Resolving the above challenges requires navigating complex materialization trade-offs involving memory usage, feature storage, and execution runtimes. Since such trade-offs might be too low-level for most ML-oriented data scientists, we design a novel "declarative" data system to handle them and let users to focus on *what* layers they want to explore rather than *how* to run the workload. We prototype our system, named Vista, in the popular integrated Spark-TensorFlow environment [6,17] to leverage these systems for orthogonal benefits such as scalability, fault tolerance, and efficient CNN inference implementation.

We formalize CNN inference operations and perform a comprehensive analysis of the memory usage behavior of this workload, in particular, using the Spark-TensorFlow environment for concreteness. We use our analysis to delineate three general dimensions of systems trade-offs for this workload. First, we compare new *logical execution plan* choices to avoid redundant CNN inference and ease memory pressure. Second, we analyze the trade-offs of key *system configuration* parameters, in particular, memory apportioning, multi-core parallelism, and data partitioning. While best practice guidelines exist for such parameters for relational workloads on systems such as Spark [2,7], novel twists in our workload necessitate deviating from such guidelines. Third, we analyze the trade-offs of two *physical execution plan* choices, viz., join operator selection and data serialization.

Unifying our above analyses, we design a simple *automated optimizer* to navigate all trade-offs and pick an end-to-end system configuration and execution plan that improves reliability and efficiency. Vista offers a simple API in Python to specify the workload and issues queries to Spark-TensorFlow under the covers. While we focus on Spark and TensorFlow due to their popularity, our work is largely orthogonal to both systems; one could replace Spark with Hadoop or a parallel RDBMS, and TensorFlow with Caffe2, CNTK, or MXNet, but still benefit from our analyses and optimization of the materialization trade-offs of this workload. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to formalize and study the materialization trade-offs of the emerging workload of large-scale feature transfer from deep CNNs for multimodal analytics over image and structured data from a systems standpoint.
- We delineate the trade-offs along the three dimensions of logical execution decisions, system configuration, and physical execution decisions. We introduce novel CNN-aware faster execution plans and explain how to optimize these three dimensions to help avoid system crashes and reduce runtimes.
- We devise a novel optimizer to handle such trade-offs automatically and build a system (named Vista) in the popular Spark-TensorFlow environment to enable data scientists to focus on their ML exploration instead of being bogged down by systems issues.
- We present an empirical evaluation of the reliability and efficiency of Vista using real-world datasets and CNNs and also analyze its handling of the trade-off space. Vista catches and avoids many crash scenarios and also reduces runtimes by 67%–90%.

Outline. The rest of this paper is organized as follows. Section 2 presents the technical background. Section 3 introduces our data model, formalizes the feature transfer workload, explains our assumptions, and provides an overview of our system. Section 4 dives into the materialization trade-offs of this workload and presents our optimizer. Section 5 presents the experimental evaluation. We discuss other related work in Section 6 and conclude in Section 7.

2. BACKGROUND

We provide some relevant technical background from both the machine learning/vision and data systems literatures.

Deep CNNs. CNNs are a type of neural networks specialized for image data [37, 52]. They exploit spatial locality of information in image pixels to construct a hierarchy of parametric feature extractors and transformers organized as layers of various types: *convolutions*, which use image filters from graphics, except with variable filter weights, to extract features; *pooling*, which subsamples features in a spatial locality-aware way; *non-linearity* to apply a non-linear function (e.g., ReLU) to all features; and *fully connected*, which is a multi-layer perceptron. A "deep" CNN just stacks such layers many times over. All parameters are trained end-to-end using backpropagation [53]. This learning-based approach to feature engineering enables CNNs to automatically construct a hierarchy of relevant image features (see Figure 1) and surpass the accuracy of prior art that relied on fixed hand-crafted features such as SIFT and HOG [32, 54].

In fact, deep CNNs have recently won numerous computer vision competitions [8, 61]. Popular deep CNN model architectures include AlexNet [47], VGG [65], Inception [67], and ResNet [38]. Our work is orthogonal to how CNNs are designed, but we note that training them from scratch incurs massive costs: they often need many GPUs for reasonable training times [3], as well as huge labeled datasets and "black magic" hyper-parameter tuning to avoid overfitting [37].

Transfer Learning with CNNs. Transfer learning is a popular paradigm to mitigate the above cost and data issues with training deep CNNs from scratch [59]. One uses a pre-trained CNN, say, ImageNet-trained AlexNet obtained from a "model zoo" [5, 10], removes its last few layers, and uses it as an image feature extractor. This "transfers" knowledge learned by AlexNet to the target prediction task. If the same CNN architecture is used and the last few layers are retrained, it is called "fine tuning," but one can also use a more interpretable model such as logistic regression for the target/downstream ML task. Such transfer learning underpins recent breakthroughs in detecting cancer [36], diabetic retinopathy [68], face recognition-based analyses [20], and multimodal recommendation algorithms combining images and structured data [56]. Generic CNN features have also been shown to beat prior hand-crafted features for many image prediction tasks [28, 35, 64, 64, 70, 71]. However, no single CNN layer is universally best for accuracy; the guideline is that the "more similar" the target task is to ImageNet, the better the higher layers will likely be [18, 27, 35, 72]. Also, lower layer features are often much larger; so, simple feature selection such as extra pooling is typically helpful [27]. Overall, data scientists have to explore at least a few CNN layers for best results with transfer learning [27, 72].

Spark and TensorFlow. Spark is a popular distributed memory-oriented and fault-tolerant data analytics system [2, 74]. At its core is the Resilient Distributed Dataset (RDD) abstraction, an immutable collection of key-value pairs that supports numerous dataflow operations, including relational operations and MapReduce. Queries compose such operations, and they are evaluated lazily. Spark uses HDFS for storage. It allows explicit caching of RDDs in distributed memory and supports disk spills during query processing. SparkSQL offers higher-level relational APIs on top of RDDs–*DataFrame* and *DataSet*–and performs RDBMS-style query optimizations [26]. Spark guidelines now recommend using *DataFrame* or *DataSet* instead of RDDs [26]. MLlib (and SparkML) is a library of popular ML algorithms implemented over Spark; it is increasingly popular for ML over structured data , especially in enterprises [4, 9].

TensorFlow (TF) is a framework for expressing ML algorithms, especially complex neural network architectures (including deep CNNs) [23, 24]. Models in TF are specified as a "computational graph," with nodes representing operations over "tensors" (multi-dimensional arrays) and edges representing data flow. To execute a graph, one selects a node to run after giving all its input data. By separating these two stages, TF uses lazy evaluation to compile the full graph and apply some optimizations. TF is under active development and has a rapidly growing user base among deep learning researchers and engineers, especially for image and text data [12]. *TensorFrames* and *SparkDL* are APIs that integrate Spark and TF [16, 17]. They enable the use of TF within Spark by invoking TF sessions from Spark workers. *TensorFrames* lets users process a *DataFrame* using TF code, while *SparkDL* offers pipelines to integrate deep neural networks into Spark queries and distribute hyper-parameter tuning. *SparkDL* is the most closely related work to Vista, since it too supports transfer learning. But unlike our work, *SparkDL* does not allow users to explore different CNN layers nor does it optimize query execution to improve reliability or efficiency. Thus, Vista could augment *SparkDL*.

3. PRELIMINARIES AND OVERVIEW

We present an example and some definitions for formalizing our data model. We then state the problem studied, explain our assumptions, and given an overview of Vista.

Example Use Case (Inspired by [56]). Consider a data scientist at an online fashion retailer working on a product recommendation system. She uses logistic regression to classify products as relevant or not for a user based on structured features such as price, brand, category, etc., and user behavior. There are also product images, which she thinks could help improve accuracy. Since building deep CNNs from scratch is too expensive for her, she uses the pre-trained deep CNN AlexNet [47] to read off the penultimate feature layer as image features. She also tries a few other layers and compares their accuracy. While this example is simplified, such use cases are growing across application domains, including online advertising (with ad images) [21], nutrition and inventory management (with food/product images) [11], and healthcare (with tissue images) [36].

Comparing multiple CNN layers is crucial for effective transfer learning [18, 27, 35, 72]. As a sanity check experiment, we took the public *Foods* dataset [11] and built an ML classifier to predict whether a particular food item is a plant based food or beverage. Using structured features (e.g., sugar and fat content) alone, a well-tuned logistic regression model yields a test accuracy of 85.2%. Including image features from *fc6* layer of ResNet raises it to 88.3%. [22].

3.1 Definitions and Data Model

We now introduce some definitions and notation to help us formalize the data model of partial CNN inference.

Definition 3.1. *A* tensor *is a multidimensional array of numbers.*[1] *The* shape *of a d-dimensional tensor* $t \in \mathbb{R}^{n_1 \times n_2 \times \ldots n_d}$ *is the d-tuple* $(n_1, \ldots n_d)$.

Definition 3.2. *A* raw image *is the (compressed) file representation of an image, e.g., JPEG. An* image tensor *is the numerical tensor representation of the image.*

Grayscale images have 2-dimensional tensors; colored ones, 3-dimensional (with RGB pixel values). We now define some abstract datatypes and functions used in this paper.

Definition 3.3. *A* TensorList *is an indexed list of tensors of potentially different shapes.*

Definition 3.4. *A* TensorOp *is a function f that takes as input a tensor t of a fixed shape and outputs a tensor $t' = f(t)$ of potentially different, but also fixed, shape. A tensor t is said to be* shape-compatible *with f iff its shape conforms to what f expects for its input.*

[1] This definition is the same as in TensorFlow [24].

DEFINITION 3.5. *A* FlattenOp *is a TensorOp whose output is a vector; given a tensor* $t \in \mathbb{R}^{n_1 \times n_2 \times \ldots n_d}$*, the output vector's length is* $\sum_{i=1}^{d} n_i$*.*

The order of the flattening is immaterial for our purposes. We are now ready to formalize the CNN model object, whose parameters (weights, activation functions, etc.) are pre-trained and fixed, as well as CNN inference operations.

DEFINITION 3.6. *A* CNN *is a TensorOp* f *that is represented as a composition of* n_l *indexed TensorOps, denoted* $f(\cdot) \equiv f_{n_l}(\ldots f_2(f_1(\cdot))\ldots)$*, wherein each TensorOp* f_i *is called a* layer *and* n_l *is the* number of layers.[2] *We use* $\hat{f}_i$ *to denote* $f_i(\ldots f_2(f_1(\cdot))\ldots)$*.*

DEFINITION 3.7. CNN inference. *Given a CNN* f *and a shape-compatible image tensor* t*,* CNN inference *is the process of computing* $f(t)$*.*

DEFINITION 3.8. Partial CNN inference. *Given a CNN* f*, layer indices* i *and* $j > i$*, and a tensor* t *that is shape-compatible with layer* f_i*,* partial CNN inference $i \rightarrow j$ *is the process of computing* $f_j(\ldots f_i(t)\ldots)$*, denoted* $\hat{f}_{i \rightarrow j}$*.*

DEFINITION 3.9. Feature layer. *Given a CNN* f*, layer index* i*, and an image tensor* t *that is shape-compatible with layer* f_i*,* feature layer l_i *is the tensor* $\hat{f}_i(t)$*.*

All major CNN layers–convolutional, pooling, non-linearity, and fully connected–are TensorOps. The above definitions capture a crucial aspect of partial CNN inference–data flowing through the layers produces a sequence of tensors. Our formalization helps us exploit this observation in VISTA to automatically optimize the execution of feature transfer workloads, which we define next.

3.2 Problem Statement and Assumptions

We are given two tables $T_{str}(\underline{ID}, X)$ and $T_{img}(\underline{ID}, I)$, where ID is the primary key (identifier), $X \in \mathbb{R}^{d_s}$ is the structured feature vector (with d_s features, including label), and I are raw images (say, as files on HDFS). We are also given a CNN f with n_l layers, a set of layer indices $L \subset [n_l]$ specific to f that are of interest for transfer learning, a downstream ML algorithm M (e.g., logistic regression), a set of system resources R (number of cores, system memory, and number of nodes). The feature transfer workload is to train M for each of the $|L|$ feature vectors obtained by concatenating X with the respective feature layers obtained by partial CNN inference. More precisely, we can state the the workload using the following set of logical queries:

$$\forall\, l \in L: \tag{1}$$

$$T'_{img,l}(\underline{ID}, g_l(\hat{f}_l(I))) \;\leftarrow\; \text{Apply } g_l \circ \hat{f}_l \text{ to } T_{img} \tag{2}$$

$$T'_l(\underline{ID}, X'_l) \;\leftarrow\; T_{str} \bowtie T'_{img,l} \tag{3}$$

$$\text{Train } M \text{ on } T'_l \text{ with } X'_l \equiv [X, g_l(\hat{f}_l(I))] \tag{4}$$

Step (2) performs partial CNN inference to materialize feature layer l and flattens it with g_l, a shape-compatible FlattenOp. Step (3) concatenates structured and image features using a key-key join. Step (4) trains M on the new multimodal feature vector. Pooling can be injected before g_l to reduce dimensionality for M [27]. The current dominant practice is to run the above queries as such, i.e., materialize all feature layers *manually* and *independently* as flat files and transfer them. Apart from being cumbersome, such an approach is inefficient due to *redundant* partial CNN inference and/or runs the risk of system crashes due to poor memory management. Our goal is to resolve these issues. *Our approach is to elevate this workload to a declarative level, obviate manual feature transfer, automatically reuse partial CNN inference results, and optimize the system configuration and execution for better reliability and efficiency.*

We make a few simplifying assumptions in this paper for tractability sake. First, we assume that f is from a roster of well-known CNNs (currently, AlexNet, VGG, and ResNet). This is a reasonable start, since most recent feature transfer applications used only such well-known CNNs from model zoos [5, 10]. We leave support for arbitrary CNNs to future work. Second, we support only one image per data example. We leave handling multiple images per example to future work. Third, we focus on using logistic regression for M, specifically, with MLlib's L-BFGS algorithm. This choice is orthogonal to this paper's focus, but it lets study CNN feature materialization trade-offs in depth. We leave support for more ML models for M (e.g., multi-layer perceptrons) to future work. Finally, we assume secondary storage is plentiful and focus on distributed memory-related issues, since storage is usually much cheaper and Spark is distributed memory-oriented and handles disk spills.

[2]For exposition sake, we focus on sequential (chain) CNNs, but it is straightforward to extend our definitions to DAG-structured CNNs such as DenseNet as well [41].

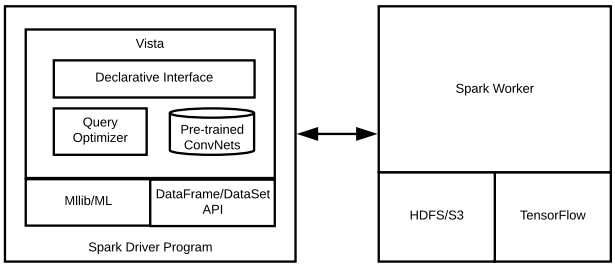


Figure 2: High-level architecture of VISTA on top of the Spark-TensorFlow combine.

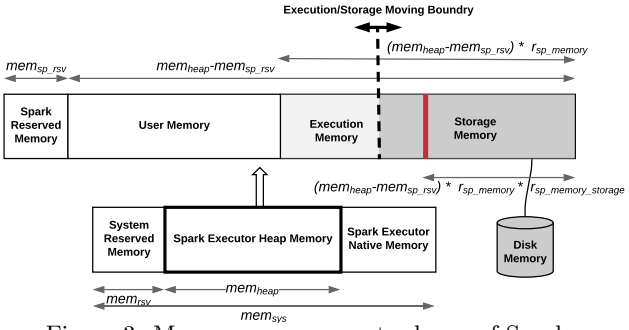


Figure 3: Memory management scheme of Spark.

3.3 System Architecture and API

We prototype VISTA as a library on top of the Spark-TensorFlow combine [6,17]. Figure 2 illustrates our system's architecture. It has four main components: (1) a "declarative" API, (2) a roster of popular named deep CNNs with named feature layers (we currently support AlexNet [47],

VGG16 [65], and ResNet50 [38]), (3) the VISTA optimizer, and (4) connectors to Spark and TensorFlow (TF). The declarative front-end API is implemented in Python; a user should specify four inputs. First is the system environment (memory, number of cores and nodes). Second is the deep CNN f and the feature layers L (from the roster) to explore for transfer learning. Third are the data tables T_{str} and T_{img}. Fourth is the downstream ML routine (with all its parameters)–currently MLlib's logistic regression.

Under the covers, VISTA uses the above inputs and invokes its optimizer (Section 4.3) to obtain a reliable and efficient combination of decisions for the logical execution plan (Section 4.2.1), key system configuration parameters (Section 4.2.2), and physical execution (Section 4.2.3). After configuring Spark accordingly, VISTA runs within the Spark Driver process to orchestrate the feature transfer task by issuing a series of queries in Spark's *DataFrame* (or *DataSet*) API [26]. VISTA uses the *TensorFrames* API [17] to invoke TF during query execution to execute our user-defined functions for (partial) CNN inference, i.e., f, $\hat{f}_l$, g_l, and $\hat{f}_{i \to j}$ for the CNNs in our roster. These functions prespecify the TF computational graphs to use, while image and feature tensors are handled using our custom *TensorList* datatype. Thus, the user does not have to write any TF code. VISTA injects the right function(s) to use during query execution based on the user's inputs and the optimizer's decisions. Finally, VISTA invokes MLlib on the joined multimodal feature representation and returns $|L|$ trained downstream models. *Overall,* VISTA *frees users from having to manually handle TF code, files of features, RDD joins, or Spark tuning for such feature transfer workloads.*

4. TRADE-OFFS AND OPTIMIZER

We first analyze the memory usage behavior of our workload. We then use our analysis to explain the trade-off space for improving reliability and efficiency. Finally, we apply our analyses to design the VISTA optimizer.

4.1 Memory Analysis of Workload

It is important to understand and optimize the memory usage behavior of our workload, since mismanaged memory can cause frustrating system crashes and/or excessive disk spills that raise runtimes in the distributed memory-based environment. We emphasize that, while the specifics of our analysis depend on the underlying system used (Spark-TF in our case), the fundamental systems-level trade-offs involved in apportioning memory between cached (intermediate) data, CNN models, and working memory for user-defined functions are largely generic and will need to be handled for any system. For concreteness sake, we explain these trade-offs using the memory management scheme of Spark (based on [15]), illustrated by Figure 3. For simplicity of exposition, we assume each worker runs one Spark Executor (a JVM process). We first explain how memory is apportioned for typical (relational) workloads on Spark. We then discuss interesting new twists in our workload that can cause crashes or inefficiency, if not handled carefully.

Overview of Spark's Memory Regions. A worker's memory is split into two main regions: System Reserved Memory for OS and other processes and Spark Executor Memory, which in turn is split into Executor Heap Memory (the maximum JVM heap size) and Executor Native Memory. In a typical relational workload on Spark, memory use is dominated by the JVM heap, while System Reserved Memory is only a few GBs. The user has to specify the maximum JVM heap size and number of cores for Executor. Spark best practices recommend using as much memory as possible for the JVM heap to reduce disk spills [7, 14, 15].[3]

The JVM heap memory is split into three sub-regions: Spark Reserved Memory Core Memory, and User Memory. The first region (typically set to 300 MB) is a safety buffer against out-of-memory errors. A fraction (typically 0.6) of the rest of the heap is Core memory, used for query processing. This is split into two sub-regions: Storage Memory, which is used for storing cached RDDs and broadcast variables and as a workspace for unrolling serialized data partitions, and Execution Memory, which is used for storing intermediate objects for RDD operations, e.g., shuffle blocks for shuffle joins and hash tables for hash aggregations. Some objects such as shuffle blocks can be spilled to disk.

The Storage Memory–Execution Memory boundary is not static. If Spark needs more of the latter, it borrows automatically from the former by *evicting* cached data partitions using an LRU cache replacement policy. Conversely, if Spark needs to cache more data, it borrows from Execution Memory. Based on the "persistence level" configured, evicted data partitions are spilled to disk or discarded (and recomputed using lineage, if needed later). But there is a maximum threshold fraction of Storage Memory (default 0.5) that is immune to eviction. Thus, Spark ensures that at least 0.6×0.5 fraction of the unreserved JVM heap is always available for caching RDDs. Finally, User Memory is used for storing objects created in user-defined transformations such as *map()* and *mapPartition()*.

Twists in Feature Transfer Workload. Spark's guidelines were designed primarily for relational workloads. But our workload requires rethinking memory management due to interesting new twists caused by deep CNNs, (partial) CNN inference, feature layers, and the downstream ML task.

First, the guideline of using most of system memory for the JVM heap no longer holds. In the Spark-TF combine, CNN inference uses Executor Native Memory *outside* the Java heap. The memory footprint of deep CNNs is nontrivial (e.g., AlexNet needed 2 GB). If the Executor uses multiple threads, each will spawn its own TF session with a replica of the CNN, multiplying the footprint. Second, many temporary objects are created for reading serialized CNNs to initialize TF sessions and for buffers to read inputs and hold feature layers created by partial CNN inference. All these go under User Memory for which Spark has almost no guidelines–it is entirely up to the user to ensure this region is large enough! But the sizes of such temporary objects depend on the number of examples in a data partition, the CNN, and L. They could vary widely, and they could be massive. For instance, Table 1 shows that *fc6* of AlexNet is of length 4096, but *conv5* of ResNet is over 400,000. Such complex memory footprint calculations will be tedious for data scientists. Third, Spark copies feature layers produced by TF into RDDs to enable MLlib to process them. Thus, Storage Memory should accommodate the new RDD(s). Fi-

[3]But if heap memory becomes extremely large, the JVM garbage collection overhead might be high. In such cases, multiple Executors per worker are recommended [7]. It is straightforward to extend VISTA to such a setting.

Table 1: Statistics of popular deep CNNs. "Layer names" is the naming convention used in the ML literature. "Output shape" is the shape of that feature layer. "MFLOPS" is the amount of computations performed by that layer's TensorOp.

AlexNet [47]			**VGG** (16 layer version) [65]			**ResNet** (50 layer version) [38]		
Layer	**Output Shape**	**MFLOPs**	**Layer**	**Output Shape**	**MFLOPs**	**Layer**	**Output Shape**	**MFLOPs**
image	[227,227,3]		image	[227,227,3]		image	[227,227,3]	
conv1	[55,55,96]	105	conv1_x	$[224, 224, 64] \times 2$	1943	conv1	[112, 112, 64]	113
conv2	[27,27,256]	224	conv2_x	$[112, 112, 128] \times 2$	2777	conv2_x	$[56, 56, 256] \times 3$	637
conv3	[13,13,384]	150	conv3_x	$[56, 56, 256] \times 3$	4626	conv3_x	$[28, 28, 512] \times 4$	980
conv4	[13,13,384]	112	conv4_x	$[28, 28, 512] \times 3$	4626	conv4_x	$[14, 14, 1024] \times 6$	1396
conv5	[13,13,256]	75	conv5_x	$[14, 14, 512] \times 3$	1323	conv5_x	$[28, 28, 512] \times 3$	771
fc6	[4096]	38	fc6	[4096]	102	fc6	[1000]	2
fc7	[4096]	17	fc7	[4096]	17			
fc8	[1000]	4	fc8	[1000]	4			

nally, for the join between the table with the feature layers and T_{str}, Execution Memory should accommodate temporary data structures created by Spark's operations, e.g., the hash table on T_{str} for broadcast join.

Memory-related Crash and Inefficiency Scenarios. The above twists give rise to various (potentially unexpected) system crash scenarios due to memory errors, as well as inefficiency issues. Having to avoid these manually could frustrate data scientists and impede ML-oriented exploration.

(1) CNN blowups. Human-readable file formats of CNNs often underestimate their in-memory footprints. Along with the replication of CNNs by multiple threads, Executor Native Memory can be easily exhausted. If users do not account for such blowups when configuring Spark, and if the blowups exceed available memory, the OS will kill the Executor.

(2) Insufficient User Memory. All Executor threads share User Memory for the CNNs and feature layer *TensorList* objects. If this region is too small due to a small overall JVM heap size or due to a large degree of parallelism, such objects might exceed available memory, leading to a crash with JVM heap out-of-memory error.

(3) Insufficient memory for Spark Driver. The Spark Driver is a JVM process that orchestrates Spark jobs among workers. In our workload, it reads and creates a serialized versions of CNNs and broadcasts them to workers. To run the downstream ML task, the Driver has to collect partial results from workers (e.g., for *collect()* and *collectAsMap()*). Without enough memory for these operations, it will crash.

(4) Very large data partitions. If a data partition is too large, Spark needs a lot of Execution Memory for RDD operations (e.g., for the join in our workload). If Execution Memory is not enough, Spark will borrow from Storage Memory by evicting cached data partitions and spilling them to disk, which wastes runtime. If even this borrowing does not suffice, it will crash with JVM heap out-of-memory error.

Overall, several execution and configuration considerations matter for reliability and efficiency. Next, we delineate these systems trade-offs precisely along three dimensions.

4.2 Dimensions of Trade-offs

The three dimensions of trade-offs we now discuss are rather orthogonal to each other, but collectively, they affect system reliability and efficiency. We explain the alternative choices for each dimension and their runtime implications.

4.2.1 Logical Execution Plan Trade-offs

Our first step is to modify the naive plan of Section 3.2 to avoid computational redundancy and reduce memory pressure. To see why redundancy exists, consider AlexNet with $L = \{fc7, fc8\}$. The naive plan, shown in Figure 4 as Plan A, performs partial CNN inference for *fc7* independently of *fc8*. As per Table 1, this implies 721 MFLOPs (49.8% of total) are redundant. A second, orthogonal, issue is *join placement*: *Should the join really come after inference?* Usually, the total size of all feature layers in L will be larger than the image tensor, e.g., even *conv5* of ResNet is thrice as large, as per Table 1. Thus, if the join is pulled below inference, as shown in Figure 4 as Plan B, shuffles costs for the join will go down. But Plan B still has the same redundancy as Plan A. The only way to remove redundancy is to break the independence of the $|L|$ queries and fuse them. This requires new TensorOps for partial CNN inference, which Vista handles by not treating CNN inference as a black box.

The first new plan we create is Plan C, "*Bulk Inference*." It materializes all feature layers of L *in one go* to avoid redundancy. The features are stored as a *TensorList* in an intermediate table and joined with T_{str}. M is then run on each feature layer (concatenated with X) projected from the *TensorList*. Plan D is a variant with the join pulled down. Empirically, we find that CNN inference operations dominate runtime (85–99% of total); thus, join placement does not matter much for runtime, but it eases memory pressure. Still, Plans C and D have high memory pressure, since they materialize all of L at once. Depending on how the memory parameters are set, this could cause crashes or a lot of disk spills, which increase runtime.

To resolve the above issues, we create a novel plan, "*Staged Inference*," Plan E in Figure 4. It splits partial CNN inference across the layers in L and invokes M on branches off of the inference path. Plan E avoids redundancy and has lower memory pressure, since feature materialization is staged out. Interestingly, Plans C/D are seldom much faster than Plan E due to a peculiarity of deep CNNs. For Plans C/D to be much faster, the CNN must "quickly" (i.e., with few layers and low MFLOPs) convert the image to small feature tensors. But such an architecture is unlikely to yield high accuracy, since it loses too much information too soon [37]. In fact, almost no popular CNN model has such an architecture. This means Plan E typically suffices from both the reliability and efficiency standpoints (we validate this in Section 5). Thus, unlike conventional optimizers that consider multiple logical plans, Vista uses only Plan E.

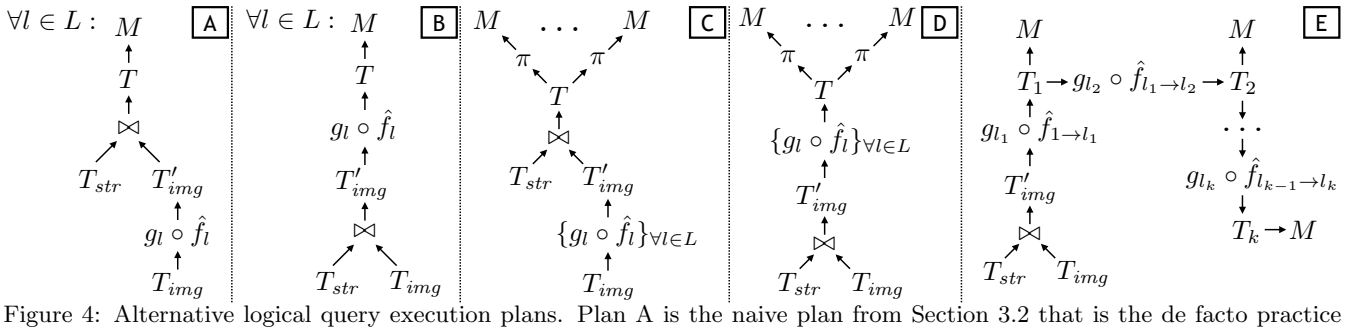


Figure 4: Alternative logical query execution plans. Plan A is the naive plan from Section 3.2 that is the de facto practice today. Plan C and D are the new "Bulk Inference" plans, while Plan E is a new "Staged Inference" plan. We define $k = |L|$.

4.2.2 System Configuration Trade-offs

Logical execution plans are generic and independent of the data system used. But as Section 4.1 explained, how the system is configured has a direct impact on reliability and efficiency. Thus, we need to understand and optimize the trade-offs of setting key system configuration parameters, in particular, the degree of parallelism in a worker, data partition sizes, and memory apportioning. Again, while these parameters are generic and matter for any system, we explain these trade-offs using the context of Spark.

We need to set number of cores per Executor (cpu_{spark}), JVM heap size (mem_{heap}), User Memory size (mem_{sp_user}), and number of data partitions (n_p). Naively, one might set cpu_{spark} to the number of cores on the node, mem_{heap} to most of system memory, and n_p to a default value that is input reader-dependent (and 200 by default for shuffles). As explained in Section 4.1, such naive settings can cause memory-related crashes or inefficiencies. But tuning these parameters manually is likely to be quite non-trivial and tedious for data scientists, since they are inter-dependent: a higher cpu_{spark} yields more parallelism for an Executor but also raises the CNN models' footprint. In turn, this means mem_{heap} should be lowered, which in turn means n_p should be raised. But if mem_{heap} is too low, Storage Memory might become too low, which causes more disk spills (especially for feature layers) and raises runtimes. Worse still, User Memory might also become too low, which could cause crashes. Lowering cpu_{spark} reduces the CNN models' footprint and allows mem_{heap} to be higher, but too low a cpu_{spark} means Spark operations become less parallel, which in turn raises runtimes, especially for the join and M. We note, however, that in the current Spark-TF combine, every TF invocation by the Executor will use all cores on the node regardless of cpu_{spark}. Nevertheless, one TF invocation per used core helps increase throughout. Finally, too low an n_p might cause crashes, while too high an n_p leads to high overhead for processing too many data partitions. Overall, one needs to navigate such non-trivial systems trade-offs that are closely tied to f, L, and M.

4.2.3 Physical Execution Trade-offs

The first decision is the physical join operator to use. Spark has two options: shuffle-hash and broadcast. In a shuffle-hash join, base tables are hashed on the join key and partitioned into "shuffle blocks" that are serialized and written to disk (for fault tolerance). Each shuffle block is then read by an assigned Executor over the network, with each Executor producing a partition of the output table using a local sort-merge join. In a broadcast join, each Executor is sent a copy of the smaller table on which it builds a hash table to join with the outer table without any shuffles. If the smaller table fits in memory, broadcast join is typically faster due to lower communication and disk spill overheads.

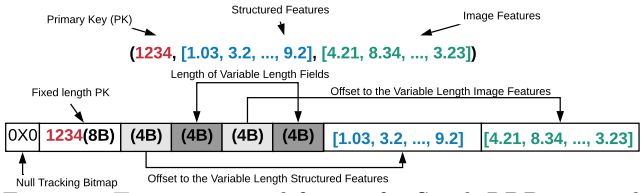


Figure 5: Tungsten record format for Spark RDDs as used in Vista for concatenated structured and image features.

The second decision is the persistence level and format for intermediate data. Since feature tensors can be much larger than raw images (see Table 1), this matters for disk spills. We first need to understand Spark's record format for intermediate RDDs; it uses an internal binary record format called the "Tungsten record format," shown in Figure 5, to avoid high Java overheads. Fixed size fields (e.g., float) use 8 B. Variable size fields (e.g., arrays) have an 8 B header with 4 B each for the offset and length of the data payload, followed by the payload in binary format. An extra bit tracks null values. Multiple such records are packed inside Java objects as binary fields to reduce garbage collection overheads. Three persistence levels are supported: in-memory deserialized or serialized (in Storage Memory), on-disk serialized, or memory-and-disk. The first level (memory only) runs the risk of the data being larger than memory, which will cause Spark to evict some data from Storage Memory. Such eviction is expensive in our workload, since it will result in CNN inference being repeated from scratch! Thus, Vista uses the third level (both memory and disk). A related decision is whether to serialize in-memory data. Compression during serialization reduces Storage Memory utilization at the cost of some extra computations. The alternative is deserialized storage. Empirically, we find that both of these options have mostly comparable runtimes (more in Section 5).

4.3 The Optimizer

We now explain how the Vista optimizer navigates the above dimensions of trade-offs automatically to improve system reliability and efficiency. Table 2 lists the notation used.

Intermediate Data Sizes. As explained in Section 4.2.1, Vista only uses Plan E. But for system configuration and physical plan decisions, Vista needs the sizes of the inter-

Table 2: Notation for Section 4 and Algorithm 1.

Symbol	Description
(A) Inputs given/ascertained from workload instance	
$\|f\|_{ser}$	Serialized size of CNN model f
$\|f\|_{mem}$	In-memory footprint of CNN model f
L	List of feature layer indices of f user wants to transfer
n_{nodes}	Number of worker nodes in cluster
mem_{sys}	Total system memory available in a worker node
cpu_{sys}	Number of cores available in a worker node
$\|T_{str}\|$	Size of the structured features table
$\|T_{img}\|$	Size of the table with images
$\|T_i\|$	Size of intermediate table T_i with feature layer $L[i]$ of f as per Figure 4(E); see Equation 5
$\|M\|$	Spark User Memory footprint of downstream model
(B) System parameters/decisions set by Vista Optimizer	
mem_{heap}	Size of Spark Executor Heap Memory (in JVM)
mem_{sp_user}	Size of Spark User Memory
mem_{sp_core}	Size of Spark Core Memory
cpu_{spark}	Number of cores assigned to Spark Executor
n_p	Number of data partitions for Spark storage
join	Physical join implementation (*shuffle* or *broadcast*)
pers	Persistence format (*serialized* or *deserailized*)
(C) Other fixed (but adjustable) system parameters	
mem_{sys_rsv}	Size of System Reserved Memory (default: 3 GB)
mem_{sp_rsv}	Size of Spark Reserved Memory (default: 0.3 GB)
c_{min}	Minimum size of Spark Core Memory as per Spark best practice guidelines (default: 2.4 GB)
p_{max}	Maximum size of data partition (default: 100 MB)
b_{max}	Maximum Spark broadcast size (default: 100 MB)
cpu_{max}	Cap recommended for cpu_{spark} (default: 8)
α_1	Fudge factor for size blowup of Spark storage data inside JVM container objects (default: 1.2)
α_2	Fudge factor for size blowup of binary feature vectors as JVM objects (default: 2)

mediate tables, $T_l, \forall l \in L$ in Figure 4(E), in the Tungsten record format. Vista estimates these automatically based on its knowledge of f. For simplicity, assume *ID* is a long integer and all features are single precision floats. Let $|X|$ denote the number of features in X. $|T_{str}|$ and $|T_{img}|$ are straightforward to calculate, since they are the base tables. For $|T_i|$ with feature layer $l = L[i]$, we have:

$$|T_i| = \alpha_1 \times (8 + 8 + 4 \times |g_l(\hat{f}_l(I))|) + |T_{str}| \quad (5)$$

Of course, Equation 5 assumes deserialized format; serialized (and compressed) data will be smaller. But these estimates suffice as safe upper bounds.

Optimizer Formalization and Simplification. The inputs for the optimizer are listed in Table 2(A). Table 2(B) lists the variables set by the optimizer. $|f|_{ser}$ and $|f|_{mem}$ are not input directly by the user; Vista has this knowledge of f in its roster. Similarly, $|M|$ is also not input directly by the user; Vista estimates it based on the specified M and the largest total number of features (based on L). For instance, for MLlib's logistic regression, $|M|$ is proportional to $(|X| + \max_{l \in L} |g_l(\hat{f}_l(I))|)$. We define two quantities to capture peak intermediate data sizes and help our optimizer set memory parameters reliably:

$$s_{single} = \max_{1 \le i \le |L|} |T_i| \quad (6)$$

$$s_{double} = \max_{1 \le i \le |L|-1} (|T_i| + |T_{i+1}|) - |T_{str}| \quad (7)$$

The ideal objective is to minimize the overall runtime subject to memory constraints. As explained in Section 4.2.2, there are two competing factors: cpu_{spark} and mem_{heap}. Raising cpu_{spark} increases parallelism, which could reduce runtimes. But it also raises the non-heap memory needed for TF, which forces mem_{heap} to be reduced, increasing potential disk spills for T_i's and raising runtimes. This tension is captured by the following objective function:

$$\min_{cpu_{spark}, n_p, mem_{sp\_core}} \frac{\tau + \max(0, \frac{s_{double}}{n_{nodes}} - 0.5 \times mem_{sp\_core})}{cpu_{spark}} \quad (8)$$

The other four variables can be set as derived variables. In the numerator, τ captures the relative total compute and communication costs, which are effectively a "constant" for this optimization. The second term captures disk spill costs for T_i's (with at least 50% of Core Memory being eviction resistant Storage Memory [7]). The denominator captures the degree of parallelism. While this objective is ideal, it is largely impractical and needlessly complicated for our purposes for three reasons. First, estimating τ is highly tedious, since it involves Spark shuffle costs, downstream model costs, etc. Second, and more importantly, we hit a point of diminishing returns with cpu_{spark} quickly, since CNN inference typically dominates total runtime and TF anyway uses all cores regardless of cpu_{spark}. That is, this workload's speedup against cpu_{spark} will be quite sublinear (confirmed by Figure 11(C) in Section 5). Empirically, we find that about 7 cores typically suffice for Spark; interestingly, a similar observation is made in Spark guidelines [14, 16]. Thus, we cap cpu_{spark} at $cpu_{max} = 8$. Third, given this cap, we can just drop the term minimizing disk spill costs, since s_{double} will typically be smaller than the total memory, even after accounting for the CNNs due to the cap. Overall, these insights yield a much simpler objective that is still a reasonable surrogate for minimizing runtimes:

$$\max_{cpu_{spark}, n_p, mem_{sp\_core}} cpu_{spark} \quad (9)$$

The constraints for the optimization are as follows:

$$1 \le cpu_{spark} \le \min\{cpu_{sys}, cpu_{max}\} - 1 \quad (10)$$

$$mem_{sp\_user} = cpu_{spark} \times \max\{|f|_{ser} + \alpha_2 \times \lceil s_{single}/n_p \rceil, |M|\} \quad (11)$$

$$mem_{heap} = mem_{sp\_user} + mem_{sp\_core} + mem_{sp\_rsv} \quad (12)$$

$$mem_{heap} + cpu_{spark} \times |f|_{mem} + mem_{sys\_rsv} < mem_{sys} \quad (13)$$

$$mem_{sp\_core} > c_{min} \quad (14)$$

$$n_p = z \times cpu_{spark} \times n_{nodes}, \text{ for some } z \in \mathbb{Z}^+ \quad (15)$$

$$\lceil s_{single}/n_p \rceil < p_{max} \quad (16)$$

Equation 10 caps cpu_{spark} and leaves a CPU for the OS. Equation 11 captures User Memory needed for reading CNN models and invoking TF, copying materialized feature layers from TF, and holding M. Equation 12 is the Executor Heap

Algorithm 1 The VISTA Optimizer Algorithm.

procedure OPTIMIZEFEATURETRANSFER:
  inputs: see Table 2(A)
  outputs: see Table 2(B)
  $cpu_{spark} \leftarrow$ `NULL`
  for $x = \min\{cpu_{sys}, cpu_{max}\} - 1$ **to** 1 **do**
    $mem'_{heap} \leftarrow mem_{sys} - mem_{sys\_rsv} - x \times |f|_{mem}$
    $n'_p \leftarrow$ NUMPARTITIONS$(s_{single}, x, n_{nodes})$
    $mem'_{sp\_user} \leftarrow x \times \max\{|f|_{ser} + \alpha_2 \times \lceil s_{single}/n'_p \rceil, |M|\}$
    $mem'_{sp\_core} \leftarrow mem'_{heap} - mem'_{sp\_user} - mem_{sp\_rsv}$
    if $mem'_{sp\_core} > c_{min}$ **then** ▷ Else, next iteration
      $cpu_{spark} \leftarrow x$ ▷ Optimal reached
      break
  if cpu_{spark} is `NULL` **then** ▷ No feasible solution
    Notify User: Insufficient System Memory
  else ▷ Optimal solution found
    $mem_{heap} \leftarrow mem_{sys} - mem_{sys\_rsv} - cpu_{spark} \times |f|_{mem}$
    $n_p \leftarrow$ NUMPARTITIONS$(s_{single}, cpu_{spark}, n_{nodes})$
    $mem_{sp\_user} \leftarrow cpu_{spark} \times \max\{|f|_{ser} + \alpha_2 \times \lceil s_{single}/n_p \rceil, |M|\}$
    $mem_{sp\_core} \leftarrow mem_{heap} - mem_{sp\_rsv} - mem_{sp\_user}$
    $join \leftarrow$ *shuffle*
    if $|T_{str}| < b_{max}$ and $\alpha_2 \times |T_{str}| \times cpu_{spark} < 0.5 \times mem_{sp\_core}$ **then**
      $join \leftarrow$ *broadcast*
    $pers \leftarrow$ *deserialized*
    if $mem_{sp\_core} < s_{double}$ **then**
      $pers \leftarrow$ *serialized*
    return $(mem_{heap}, mem_{sp\_user}, mem_{sp\_core}, cpu_{spark}, n_p, join, pers)$

procedure NUMPARTITIONS$(s_{single}, x, n_{nodes})$:
  $totalcores \leftarrow x \times n_{nodes}$
  return $\lceil \frac{s_{single}}{p_{max} \times totalcores} \rceil \times totalcores$

Memory definition (Figure 3). Equation 13 constrains the total memory as per Figure 3; $cpu_{spark} \times |f|_{mem}$ is the Executor Native Memory used by TF. Equation 14 captures a Spark guideline for Core Memory [7]. Equation 15 requires n_p to be a multiple of the number of worker processes to avoid skews, while Equation 16 bounds the size of an intermedaite data partition, as per Spark guidelines [1].

Optimizer Algorithm. Due to our above observations, the algorithm is simple: linear search on cpu_{spark} to satisfy all constraints. Algorithm 1 presents it formally. If cpu_{spark} is still `NULL` after the search, there is no feasible solution, i.e., the system memory is too small to satisfy some constraints (say, Equation 14 or 12). In this case, VISTA notifies the user accordingly, and the user can provision machines with more memory. If cpu_{spark} is not `NULL`, we have the optimal solution. The other variables are set based on the constraints. We set *join* to *broadcast* if the maximum broadcast data size constraint is satisfied and Execution Memory is sufficient; otherwise, we set it to *shuffle*. Finally, as per Section 4.2.3, *pers* is set to *serialized*, if disk spills are likely (based on the newly set mem_{sp_core}). This is a bit conservative, since not all pairs of intermediate tables might spill, but empirically, we find that this conservatism does not affect runtimes significantly (more in Section 5). We leave more complex optimization criteria to future work.

5. EXPERIMENTAL EVALUATION

We empirically validate if VISTA is able to improve reliability and efficiency of feature transfer workloads. We then drill into how it handles the trade-off space.

Datasets. We use two real-world datasets: *Foods* [11] and *Amazon* [55]. *Foods* has about 20,000 examples with 130 structured numeric features such as nutrition facts along with pairwise/ternary feature interactions and an image of each food item. The target represents if the food item is plant-based or not. *Amazon* is larger, with about 200,000 examples with structured features such as price, title, and list of categories, as well as a product image. The target represents the sales rank, which we binarize as a popular product or not. We pre-processed title strings to extract 100 numeric features (an "embedding") using the popular Doc2Vec procedure [51]. We convert the indicator vector for categories into 100 numeric features using PCA. All images are resized to 227×227 resolution, as required by most popular CNNs. All of our data pre-processing scripts and system code are available on the project webpage: `https://adalabucsd.github.io/vista.html`. We hope our efforts help spur more research on this topic.

Workloads. We use three popular ImageNet-trained deep CNNs: AlexNet [47], VGG16 [65], and ResNet50 [38], obtained from [5,10]. They complement each other in terms of model size and total MFLOPs [29]. We select the following interesting layers for feature transfer from each (see Table 1 for layer sizes): *conv5* to *fc8* from AlexNet ($|L| = 4$); *fc6* to *fc8* from VGG ($|L| = 3$), and top 5 layers from ResNet (from its last two layer blocks [38]), with only the topmost layer being fully-connected. Following standard practices [18,72], we apply max pooling on the convolutional feature layers to reduce their dimensionality before using them for M.[4] As for M, we run MLlib's logistic regression for 10 iterations.

Experimental Setup. We use a cluster with 8 workers and 1 master in an OpenStack instance on CloudLab, a free and flexible cloud resource for research [60]. Each node has 32 GB RAM, 8 Intel Xeon @ 2.00GHz CPUs, and 300 GB Seagate Constellation ST91000640NS HDDs. They run Ubuntu 16.04. We use Apache Spark v2.2.0 with *TensorFrames* v0.2.9 integrating it with TensorFlow v1.3.0. Spark runs in standalone mode. Each worker runs one Executor. HDFS replication factor is three; input data is ingested to HDFS and read from there. Each runtime reported is the average of three runs with 90% confidence intervals. Note that our work is orthogonal to whether one uses CPUs or GPUs for CNN inference. GPUs will reduce absolute CNN inference runtimes across the board, but the trade-offs and relative trends we study will remain the same.

5.1 End-to-End Reliability and Efficiency

We compare VISTA with four baselines: two naive and two strong. *Naive-1* (1 CPU per Executor) and *Naive-5* (5 CPUs per Executor) represent the current dominant practice of running all feature transfer queries separately (Section 3.2), with Spark configured by best practices [7, 14] (29 GB JVM heap, memory-and-disk deserialized, shuffle join, and defaults for all other parameters, including n_p and heap memory divisions). *Naive-5 with Pre-Mat* and *Bulk*

[4]The filter width and stride for max pooling are set to reduce the feature tensor to a 2×2 grid of the same depth.

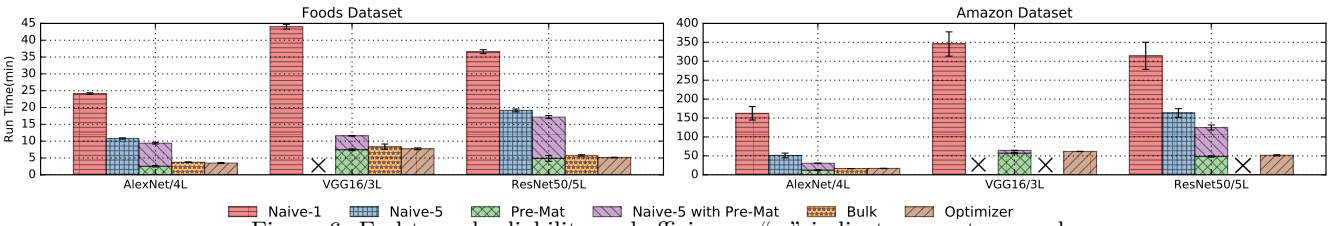


Figure 6: End-to-end reliability and efficiency. "×" indicates a system crash.

are strong baselines based on our analysis of the logical plan trade-offs (Section 4.2.1). In *Naive-5 with Pre-Mat*, the lowest feature layer (e.g., *conv5* for AlexNet) is materialized beforehand and used in place of raw images for all subsequent CNN inference. *Bulk* is the new Bulk Inference plan explained in Section 4.2.1 (with 5 CPUs per Executor). Vista is our optimizer-picked plan, including for system configuration (Section 4.3). Note that *Naive-5 with Pre-Mat* and *Bulk* actually need parts of the Vista code base. Figure 6 presents the results.

We see that Vista improves reliability and/or efficiency across the board. *Naive-5* crashes on both datasets with VGG16; *Bulk* crashes on *Amazon* with VGG16 and ResNet50. These are due to memory pressures caused by CNN model blowups or User Memory blowups (Section 4.1). When *Bulk* does not crash, its efficiency is comparable to Vista, which validates our analysis in Section 4.2.1. *Naive-5 with Pre-Mat* does not crash, but its efficiency is comparable to *Naive-5* and worse than Vista. This is because the feature layers of AlexNet and ResNet are much larger than the raw images, which raises data I/O and join costs (we provide the runtime breakdowns in the technical report [22]). Compared to *Naive-5*, Vista is 67%–73% faster; compared to *Naive-1*, 82%–90%. These gains arise because Vista removes redundancy in partial CNN inference and reduces disk spills. Of course, the exact gains depend on the CNN and L: if more of the higher layers are explored, the more redundancy there is and the faster Vista will be. Overall, Vista never crashes and offers the best (or near-best) efficiency on these workloads. This confirms the benefits of an automatic optimizer such as ours for improving reliability and efficiency, which could reduce both user frustration and costs.

5.2 Drill-Down Analysis of Trade-offs

We now drill into the various dimensions of trade-offs discussed in Section 4 to validate if Vista navigates such trade-offs appropriately. For this subsection, we use the less resource-intensive *Foods* dataset, but alter it "semi-synthetically" for some experiments to analyze Vista performance in new operating points. In particular, when specified, we vary the data scale (by replicating tuples and denoted, e.g., as "4X") or the number of structured features (with random values). For the sake of uniformity, unless specified otherwise, we use all 8 workers, fix cpu_{spark} to 4, and fix Core Memory to be 60% of the JVM heap. We set the other parameters as per the Vista optimizer. The layers explored for each CNN are the same as before.

Intermediate Table Sizes. We check if our estimates of s_{single} are accurate. While Vista only uses Staged inference, we include Bulk inference for a comparison. Figure 7 shows the estimated and actual sizes. We see that the estimates are accurate for the deserialized in-memory

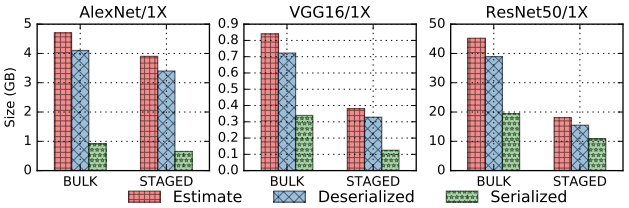


Figure 7: Size of largest intermediate table.

data, with a reasonable safety margin. Interestingly, Bulk is not that much larger than Staged for AlexNet. This is because among its four layers explored, *conv5* is disproportionately large, while for the other two, the layer sizes are more comparable (see Table 1). Serialized is obviously smaller than deserialized, since Spark compresses the data. Interestingly, AlexNet feature layers seem more compressible; we verified that its features had many zero values (caused by ReLU). On average, AlexNet features had only 13.0% nonzero values, while VGG16's and ResNet50's had 36.1% and 35.7%, respectively. Exploiting such compression gains fully in Vista presents an interesting avenue for future work.

Logical Plan Decisions. We compare four combinations: Bulk or Staged Inference combined with inference after join ("AJ") or before join ("BJ"). We vary both $|L|$ (by dropping successive lower layers) and data scale for AlexNet and ResNet. Figure 8 shows the results. We see that the runtime differences between all plans are insignificant for low data scales or low $|L|$ on both CNNs. But as $|L|$ or the data scale goes up, both Bulk plans get much slower, especially for ResNet (Figure 8(B,D)), since they face more disk spills for the massive intermediate table generated. Across the board, the AJ plans are mostly comparable to their BJ counterparts, but marginally faster at larger scales. These results validate our choice in Vista to only use the Staged/AJ combination (called Plan E in Figure 8 and Section 4.2.1).

Physical Plan Decisions. We compare four combinations: shuffle ("SHUFFLE") or broadcast ("BROAD") join and serialized ("SER") or deserialized ("DESER") persistence format. We vary data scale and number of structured features ($|X_{str}|$) for both AlexNet and ResNet. The logical plan used is Staged/AJ. Figure 8 shows the results. We see that all four plans are almost indistinguishable regardless of the data scale for ResNet (Figure 9(B)), except at the 8X scale, the SER plans outperform the DESER plans. For AlexNet, BROAD plans outperform SHUFFLE plans (Figure 9(A)). Figure 9(C) shows this gap remains as $|X_{str}|$ increases, with the BROAD plans eventually crashing. For ResNet, however, Figure 9(D) shows that both SER plans are slightly faster than their DESER counterparts, but the BROAD plans still crash eventually. The gap between SER and DESER is significant for ResNet (but not AlexNet),

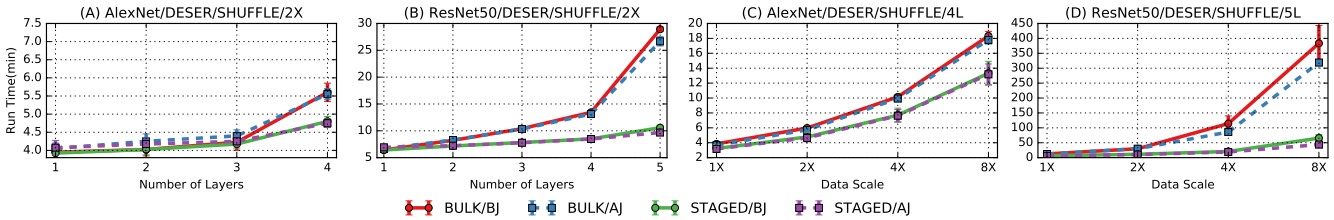


Figure 8: Runtime comparison of logical plan decisions for varying data scale and number of feature layers explored.

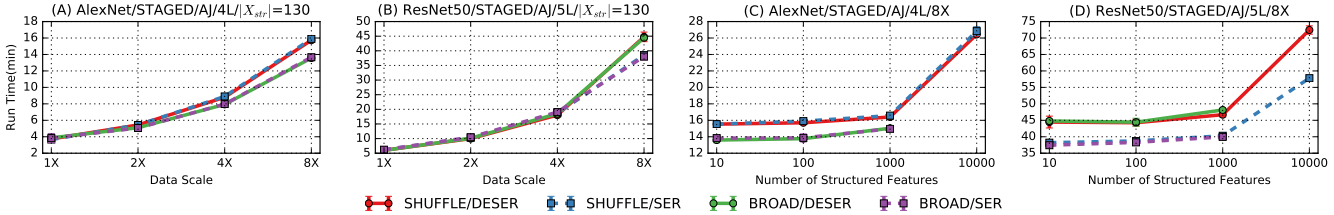


Figure 9: Runtime comparison of physical plan decisions for varying data scale and number of structured features.

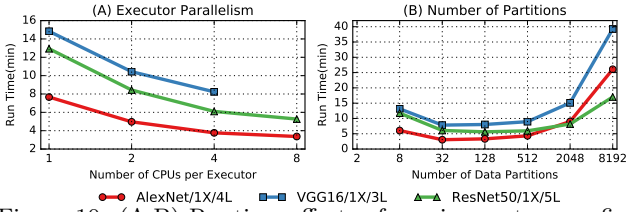


Figure 10: (A,B) Runtime effects of varying system configuration parameters. Logical and Physical plan choices are fixed to STAGED/AJ/SHUFFLE/DESER.

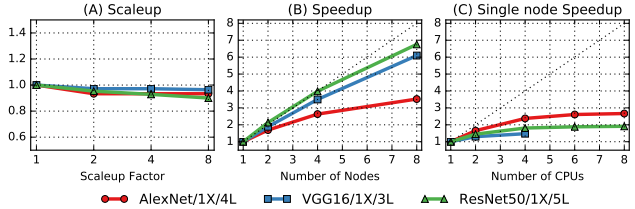


Figure 11: (A,B) Scaleup and speedup behavior on cluster. (C) Speedup ratio for varying cpu_{spark} on one node with 0.25x data. Logical and Physical plan choices are fixed to STAGED/AJ/SHUFFLE/DESER.

since at 8X scale, its largest intermediate table requires disk spills. Our optimizer handles such trade-offs automatically.

System Configuration Decisions. We vary cpu_{spark} and n_p, with the optimizer setting the memory parameters accordingly. The plan is STAGED/AJ/SHUFFLE/DESER. Figures 10(A,B) show the results. As explained in Section 4.3, the runtime decreases with cpu_{spark} for all CNNs, but VGG eventually crashes (at 8 cores) due to Native Memory blowup caused by CNN replicas. The runtime decrease with cpu_{spark} is, however, sub-linear. To drill into this issue, we plot the speedup against cpu_{spark} on 1 node for data scale $0.25X$ (to avoid disk spills). Figure 11(C) shows the results: the speedups plateau at about 4 cores. As mentioned in Section 4.3, this is to be expected, since CNN inference dominates total runtime and TF always uses all cores regardless of cpu_{spark}. Due to space constraints, we provide the runtime breakdowns in the technical report [22]. We also varied the JVM heap size; the runtimes did not change too much, unless the intermediate table is spilled to disk. The deatils are presented in the technical report [22].

Figure 10(B) shows non-monotonic behaviors with n_p. At very low n_p, Spark crashes due to insufficient Core Memory for the join. As n_p goes up, runtimes go down, since Spark exploits more of the parallelism available (up to 32 cores usable). But eventually, runtimes rise again due to Spark overheads for handling too many tasks. In fact, when $n_p > 2000$, Spark compresses the task statuses sent to the master, which increases overhead substantially. By setting n_p automatically, VISTA frees data scientists from having to navigate such trade-offs. Our optimizer sets n_p at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, all of which yield runtimes close to the fastest for varying n_p.

Scalability. Finally, we evaluate the speedup (strong scaling) and scaleup (weak scaling) of the plan STAGED/AJ/SHUFFLE/DESER as we vary the number of worker nodes (and also data scale for scaleup). While partial CNN inference and M are embarassingly parallel, data reads from HDFS and the join can bottleneck scaling behaviors. Figures 11 (A,B) show the results. We see near linear scaleup with all 3 CNNs. But Figure 11(B) shows that the AlexNet sees a markedly sub-linear speedup, while VGG and ResNet exhibit near-linear speedups. To explain this gap, we drilled into the Spark logs and obtained the time breakdown for data reads and CNN inference coupled with the first iteration of logistic regression for each layer. For all 3 CNNs, data reads exhibit sub-linear speedups due to the notorious "small files" problem of HDFS with the images [13]. But for AlexNet in particular, even the second part is sub-linear, since its absolute compute time is much lower than that of VGG or ResNet. So, Spark overheads become non-trivial in AlexNet's case. Due to space constraints, we provide more analysis of the speedups in the technical report [22].

Summary of Experimental Results. Overall, ignoring the interconnected trade-offs of logical execution plan, system configuration, and physical execution plan often raises runtimes (even by 10x) or causes crashes. Staged Inference significantly outperforms both naive execution and Bulk Inference at large scales. Pulling partial CNN inference above the join does not affect efficiency significantly but eases memory pressure. Proper system configuration for memory appor-

tioning, data partitioning, and parallelism in a CNN- and feature layer-aware manner is crucial for reliability and efficiency. If the structured dataset is small, broadcast join marginally outperforms shuffle join but causes crashes at larger scales. Serialized disk spills are comparable to deserialized but marginally better in some cases. Overall, Vista manages and optimizes these trade-offs automatically, thus freeing data scientists to focus on ML-related exploration. We also found that at least one feature layer lifts downstream accuracy significantly in each workload, but this is orthogonal to our focus; we present the accuracy comparison in our technical report due to space constraints [22].

5.3 Discussion and Limitations

TF is a powerful tool for building deep learning models but has poor support for data independence and structured data management, which forces users to manually manage data files, distribution, memory, etc. On the other hand, Spark has much better data independence, but poor native support for deep learning. Thus, the Spark-TF combine is a powerful marriage of two complementary systems for unified analytics over structured and unstructured data. But as our work shows, this combine is still rudimentary and much work is still needed to improve system reliability, efficiency, and user productivity. Vista is a first step in this direction.

We recap key assumptions and limitations of this work. Vista supports and optimizes large-scale feature transfer from deep CNNs for multimodal analytics combining structured data with images (one image per example). It supports a roster of popular CNNs for transferring features and MLlib's logistic regression for downstream ML. We did not consider secondary storage space a major concern. Nothing in Vista makes it difficult to relax these assumptions. For instance, supporting more downstream MLlib models mainly requires their memory footprints, while supporting arbitrary CNNs requires static analysis of TF computational graphs. We leave such extensions to future work.

6. OTHER RELATED WORK

Multimodal Analytics. Transfer learning is used for other multimodal analytics tasks too, including image captioning [45]. Our focus is on systems for integrating images with structured features. A related but orthogonal line of work is "multimodal learning" in which deep neural networks (or other models) are trained from scratch on multimodal data [58, 66]. While feasible for some applications, this approach faces the same cost and data issues of training deep CNNs from scratch, which transfer learning mitigates.

Multimedia DBMSs. There is prior work in the database and multimedia literatures on DBMSs for "content-based" image retrieval (CBIR), video retrieval, and other queries over multimedia data [25, 43]. They relied on older handcrafted features such as SIFT and HOG [32, 54], not learned or hierarchical CNN features, although there is a resurgence of interest in CBIR with CNN features [70,73]. Such systems are orthogonal to our work, since we focus on feature transfer with deep CNNs for multimodal analytics, not CBIR or multimedia queries. One could integrate Vista with multimedia DBMSs. NoScope is a system to detect objects in video streams using deep CNNs [44]. It reduces costs by building a "cascade" of faster models that exploit temporal and other redundancy in video. Vista is orthogonal, since it focuses on feature transfer, not cascades. One could combine these systems for more efficiency gains.

Query Optimization. Our work is inspired by a long line of work on optimizing SQL queries with UDFs, multi-query optimization (MQO), and self-tuning DBMSs. For instance, [30, 39] studied the problem of predicate migration for optimizing complex relational queries with joins and UDF-based predicates. Also related is [19], which studied "semantic" optimization of queries with predicates based on data mining classifiers. Unlike such works on queries with UDFs in the `WHERE` clause, our work can be viewed as optimizing UDFs expressed using TensorFlow in the `SELECT` clause for materializing CNN feature layers. We study and optimize novel materialization trade-offs for this workload. The new plans of Vista can be viewed as a form of MQO, which has been studied extensively for SQL queries [63]. Vista is the first system to apply the general idea of MQO to complex CNN feature transfer workloads by formalizing partial CNN inference operations as first-class citizens for query processing and optimization. Vista can also be viewed as a model selection management system [48] that optimizes for CNN-based feature engineering. In doing so, our work expands a recent line of work on materialization optimizations for feature subset selection in linear models [46,76] and integrating ML with relational joins [31, 49, 50, 62]. Finally, there is much prior work on auto-tuning system configuration for relational and MapReduce workloads (e.g., [40,69]). Our work is inspired by such systems, but we focus specifically on the emerging workload of large-scale CNN feature transfer and study its novel twists and trade-offs in depth.

7. CONCLUSIONS AND FUTURE WORK

The success of deep CNNs presents exciting new opportunities for exploiting images and other unstructured data sources in data-driven applications that have hitherto relied mainly on structured data. But realizing the full potential of this integration requires data analytics systems to evolve and elevate CNNs as first-class citizens for query processing, optimization, and system resource management. In this work, we take a first step in this direction by building upon the Spark-TensorFlow combine to support and optimize a key emerging workload in this context: feature transfer from deep CNNs for multimodal analytics. By enabling more declarative specification and by formalizing partial CNN inference, Vista automates much of the data management-oriented complexity of this workload, thus improving system reliability and efficiency, which in turn reduces resource costs and potentially improves data scientist productivity.

As for future work, we plan to support more general forms of CNNs and downstream ML tasks, as well as the interpretability of such models in data analytics. We also plan to deepen the integration of deep learning models with data analytics systems to enable seamless and efficient type-agnostic multimodal data analytics involving other types of unstructured data as well, a vision we call "database perception."

8. REFERENCES

[1] Adaptive execution in spark. `https://issues.apache.org/jira/browse/SPARK-9850`. Accessed January 31, 2018.

[2] Apache spark: Lightning-fast cluster computing. http://spark.apache.org. Accessed January 31, 2018.

[3] Benchmarks for popular cnn models. https://github.com/jcjohnson/cnn-benchmarks. Accessed January 31, 2018.

[4] Big data analytics market survey summary. https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/#4b513fce39a1. Accessed January 31, 2018.

[5] Cafee model zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo. Accessed January 31, 2018.

[6] Deep learning with apache spark and tensorflow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html. Accessed January 31, 2018.

[7] Distribution of executors, cores and memory for a spark application running in yarn. https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application. Accessed January 31, 2018.

[8] History of computer vision contests won by deep cnns. http://people.idsia.ch/~juergen/computer-vision-contests-won-by-gpu-cnns.html. Accessed January 31, 2018.

[9] Kaggle survey: The state of data science and ml. https://www.kaggle.com/surveys/2017. Accessed January 31, 2018.

[10] Models and examples built with tensorflow. https://github.com/tensorflow/models. Accessed January 31, 2018.

[11] Open food facts dataset. https://world.openfoodfacts.org/. Accessed January 31, 2018.

[12] A peek at trends in machine learning by andrej karpathy. https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106. Accessed January 31, 2018.

[13] The small files problem of hdfs. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed January 31, 2018.

[14] Spark best practices. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Accessed January 31, 2018.

[15] Spark memory management. https://0x0fff.com/spark-memory-management/. Accessed January 31, 2018.

[16] Sparkdl: Deep learning pipelines for apache spark. https://github.com/databricks/spark-deep-learning. Accessed January 31, 2018.

[17] Tensorframes: Tensorflow wrapper for dataframes on apache spark. https://github.com/databricks/tensorframes. Accessed January 31, 2018.

[18] Transfer learning with cnns for visual recognition. http://cs231n.github.io/transfer-learning/. Accessed January 31, 2018.

[19] Efficient evaluation of queries with mining predicates. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 529–. IEEE Computer Society, 2002.

[20] Deep neural networks are more accurate than humans at detecting sexual orientation from facial images, 2017.

[21] Personal communication with google ads infrastructure, 2017.

[22] Materialization trade-offs for feature transfer from deep cnns for multimodal data analytics [technical report], Jan. 2018. https://adalabucsd.github.io/papers/TR_2018_Vista.pdf.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org; accessed December 31, 2017.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283. USENIX Association, 2016.

[25] D. A. Adjeroh and K. C. Nwosu. Multimedia database management-requirements and issues. *IEEE MultiMedia*, 4(3):24–33, Jul 1997.

[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[27] H. Azizpour, A. S. Razavian, J. Sullivan, A. Maki, and S. Carlsson. Factors of transferability for a generic convnet representation. *IEEE transactions on pattern analysis and machine intelligence*, 38(9):1790–1802, 2016.

[28] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky. Neural codes for image retrieval. In *European conference on computer vision*, pages 584–599. Springer, 2014.

[29] A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[30] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, June 1999.

[31] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proc. VLDB Endow.*, 10(11):1214–1225, Aug. 2017.

[32] N. Dalal and B. Triggs. Histograms of oriented

gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893. IEEE Computer Society, 2005.

[33] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[35] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 647–655, Bejing, China, 22–24 Jun 2014. PMLR.

[36] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, Jan. 2017.

[37] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

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

[39] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 267–276. ACM, 1993.

[40] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[41] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[42] Y. Jing, D. Liu, D. Kislyuk, A. Zhai, J. Xu, J. Donahue, and S. Tavel. Visual search at pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1889–1898. ACM, 2015.

[43] O. Kalipsiz. Multimedia databases. In *IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 111–115, 2000.

[44] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Optimizing deep cnn-based queries over video streams at scale. *CoRR*, abs/1703.02529, 2017.

[45] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):664–676, Apr. 2017.

[46] P. Konda, A. Kumar, C. Ré, and V. Sashikanth. Feature selection in enterprise analytics: A demonstration using an r-based data analytics system. *Proc. VLDB Endow.*, 6(12):1306–1309, Aug. 2013.

[47] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[48] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[49] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1969–1984. ACM, 2015.

[50] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: Towards optimization across linear and relational algebra. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 1:1–1:4. ACM, 2016.

[51] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[52] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[53] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[54] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

[55] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52. ACM, 2015.

[56] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52. ACM, 2015.

[57] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[58] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng. Multimodal deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 689–696, USA, 2011. Omnipress.

[59] S. J. Pan and Q. Yang. A survey on transfer learning.

IEEE Transactions on knowledge and data engineering, 22(10):1345–1359, 2010.

[60] R. Ricci and E. Eide. Introducing cloudlab: Scientific infrastructure for advancing cloud architecturesand applications. *; login:*, 39(6):36–38, 2014.

[61] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[62] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 3–18. ACM, 2016.

[63] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[64] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

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

[66] N. Srivastava and R. Salakhutdinov. Multimodal learning with deep boltzmann machines. volume 15, pages 2949–2980. JMLR.org, Jan. 2014.

[67] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[68] G. V, P. L, C. M, and et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*, 316(22):2402–2410, 2016.

[69] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[70] J. Wan, D. Wang, S. C. H. Hoi, P. Wu, J. Zhu, Y. Zhang, and J. Li. Deep learning for content-based image retrieval: A comprehensive study. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 157–166. ACM, 2014.

[71] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3360–3367. IEEE, 2010.

[72] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3320–3328. MIT Press, 2014.

[73] J. Yue-Hei Ng, F. Yang, and L. S. Davis. Exploiting local features from deep networks for image retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 53–61, 2015.

[74] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[75] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[76] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 265–276, 2014.

APPENDIX

A. PRE MATERIALIZING A BASE LAYER

In practice a data scientist would not explore all the layers in a CNN. Most of the lower layers are learned to distinguish low level features which are common in all images, hence has less discriminating power. So a natural inclination would be to pre-materialize features from a base layer (say conv5 from AlexNet) which can be later use to explore other top level layers (say fc6, fc7 etc...) without doing the CNN inference all the way from raw images. This can drastically reduce the number of computations required for the CNN inference (computing AlexNet conv5 features takes 92% of total computations required by AlexNet for CNN inference) and one would expect similar runtime reductions.

However the stored CNN feature sizes are generally larger than the compressed image formats such JPEG (specially features from conv layers) and this not only increases the secondary storage requirements but also the IO cost of the CNN feature transfer workload both when initially reading from the disk and at join time when sending over the network. For AlexNet the size of the stored image features from the 4^{th} layer from the top (conv5) in [5]Parquet format is ∼3 times larger than the raw images in the **Foods** dataset and for ResNet50 5^{th} layer from top (conv_4_6) is ∼44 times larger than the size of the raw images(see Table. 3).

Table 3: Sizes of pre-materialized feature layers for **Foods** dataset(size of raw images is 0.26 GB).

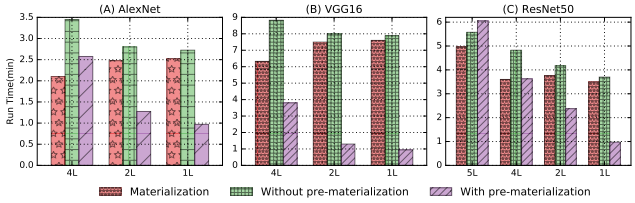
	Materialized Layer Size (GB) (layer index starts from the last layer)			
	1^{st}	2^{nd}	4^{th}	5^{th}
AlexNet	0.08	0.14	0.72	
VGG16	0.08	0.20	1.19	
ResNet50	0.08	2.65	3.45	11.51

We perform a new set of experiments exploring different number of layers (say 4, 2, and 1 layers from the top separately) for each CNN model where the lowest layer is pre-materialized. Therefore for evaluating the downstream ML model for the lowest layer no CNN inference is required. For the other layers the CNN inference is done iteratively starting from the pre-materialized features (instead of raw images) using the `STAGED` and `AJ` logical plan. Experimental set up is same as in Section. 5.2 and the persistence and join operator is set to `DESER` and `SHUFFLE` respectively.

For AlexNet and VGG16 when exploring top 4, 2, and 1 layers the materialization time of the features of the lowest layer increases as evaluating higher layer requires more computations (see Figure. 12 (A) and (B)). However for ResNet50 there is a sudden drop in the materialization time of 5^{th} layer features to materialization time of 4^{th} layer features. This can be attributed to the disk IO overhead of writing out 5^{th} layer image features which is ∼3 times larger than that of layer 4 (see Figure. 12 (C)). Surprisingly we found that starting from a pre-materialized feature layer instead of raw images may or may not decrease the overall CNN feature transfer workload runtime. For AlexNet and VGG16, plan which starts from the pre-materialized base layer improves the runtime compared to the plan which starts from raw images. The time reduction is high when materializing last layers (1^{st} and 2^{nd} layers from the top) where the size of the materialized features is small. But for ResNet50 this is not always the case. When performing CNN feature transfer for the top 5 layers, plan which starts from the pre-materialized layer performs worse than the plan which starts with raw images. This is because of the high IO overhead of reading large image features and sending them over the network (for the shuffle join) which is ∼44 times the size of raw images (see Figure. 3 (C)). In this case the computational time saved by materializing features from a base layer is dwarfed by the high IO overhead of handling large image features. Hence materializing CNN features from a base layer may not be advantageous always.

[5]https://parquet.apache.org/

Figure 12: Runtimes when using a pre-materialized features from a base layer

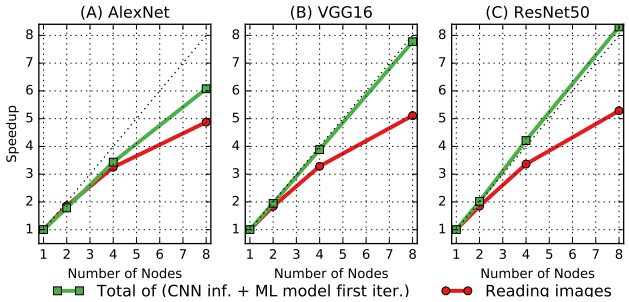


B. RUNTIME BREAKDOWN

In Figure. 10 (D) we see than all CNN models show sub-linear speedup behavior and speedup behavior of AlexNet is worse than other two models. However CNN inference is an embracingly parallel task and one would expect near linear speedups for CNN feature transfer workloads. To explain this peculiarity we drill-down into the time breakdowns of the workloads and explore where the bottlenecks occur. Typically Spark operations are run in pipelined fashion (except `mapPartition`, and `shuffling` operations). Therefore taking a precise time breakdown for all the sub-tasks is not possible. However Spark breaks down tasks into multiple stages when ever there is a shuffle boundary (e.g. a shuffle join) or into separate jobs when ever a Spark action (e.g `collect`, `count`) is being called. Therefore in the CNN feature transfer workloads that we explore reading of input data (structured data file and image files) and writing of shuffle-blocks will be separated into two sub stages. Also every iteration in the Logistic Regression (LR) model (downstream ML task) will also invoke a new job and the time consumed by each job can be obtained from the Spark Admin UI.

In the downstream Logistic Regression model, the time spent for training the model on features from a specific layer is dominated by the runtime of the first iteration. In the first iteration partial CNN inference has to be done starting either from raw images or from the image features from the layer below and the later iterations will be operating on top of the already materialized features. For the input reading the time is dominated by image data, because there are large number of small files compared to the structured file which is a one large file [13]. Table. 4 summarizes the time breakdown for the CNN feature transfer workloads when using different CNN models with the Foods Dataset. It can be seen that most of the time is spent on performing the CNN inference and LR 1^{st} iteration on the first layer (e.g 5^{th} layer from top for ResNet50) where the CNN inference has to be performed starting from raw images. Also it can

Figure 13: Drill-down analysis of Speedup Curves



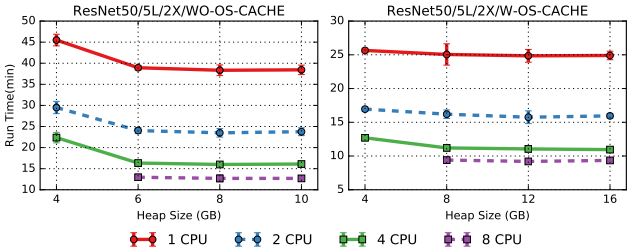
be seen that time consumed by ResNet50 and VGG16 is much higher than the AlexNet model.

We also analyze the speedup behavior for the input image reading and the sum of CNN inference and LR 1^{st} iteration times (see Figure. 13). Recall that for the overall speedup curves in Figure. 10 we observer sub-linear speedups for all three CNN models. However when we separate out the CNN inference plus LR 1^{st} iteration time we see slightly super linear speedups for ResNet50, near linear speedups for VGG16 and slightly better sub-linear speedups for AlexNet. Compared to AlexNet, VGG16 and ResNet50 CNN inferences are highly compute intensive (AlexNet, ResNet50 and VGG16 requires 725 MFLOPS, 7,754 MFLOPS and 18,721 MFLOPS respectively) and therefore shows better speedup behavior. For ResNet50 the intermediate data sizes are large and when we increase the number of nodes the amount of disk spills decreases. Hence we see slightly super linear speedups for ResNet. For reading input images we see sub-linear speedups which is bottlenecked by HDFS.

C. HEAP SIZE & CPUS PER EXECUTOR

Both Heap Size and CPUs per Executor are important factors for reducing the CNN feature transfer workload runtime. When increasing the CPUs per Executor the time spent on CNN inference and downstream ML model will decrease due to increased parallelism. However TensorFlow is using all the cores available in the machine and therefore time spent on CNN inference will not reduce proportionally when increasing the Executor parallelism. Having a large Heap size can help reducing the disk spills and thereby reduce the runtime. We experiment the effect of these two factors on CNN feature transfer workload runtime by fixing the Executor parallelism and changing the Heap size. The experimental setup is similar to that of Section. 5.2. To remove the effect of OS page cache playing a role on the runtimes we periodically flush the page caches (`WO-OS-CACHE`) while the workload is running. We see that both when increasing the Executor parallelism and Heap size the runtime decreases (see Figure. 14 (A)). But increasing the Executor parallelism contributes more to the runtime reduction even though it eventually plateaus. Also increasing the Heap size beyond a certain limit when there are no more disk spills does not contribute to significant runtime reductions. When we repeat the same experiment without flushing the OS page cache periodically (`W-OS-CACHE`) we see that the runtimes are largely agnostic to the Heap size (see Figure. 14 (B)). This is because even though the Spark Storage Memory is small, OS is caching the disk blocks in memory and thereby shows similar performance to having a large Storage Memory.

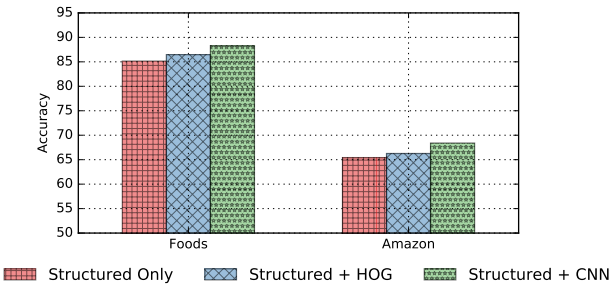
Figure 14: Changing Heap Size and CPUs per Executor (Workload crashes when running 8 CPUs per Executor with 4 GB of Heap Memory)



D. ACCURACY

In this section we verify that using CNN features as image representations can improve downstream ML model accuracy. For both Foods and a sample of Amazon (20,000 records) datasets we evaluate the downstream logistic regression model accuracy with (1) only using structured features and (2) structured features combined with image features from different layers of AlexNet and ResNet models. For the Foods dataset, a well tuned logistic regression model (with $||\ell||_1$ regularization) achieves a test accuracy of 85.2% with using only structured features. Using images features from the $fc8$ layer of AlexNet raises it to 86.2% and $fc6$ layer of ResNet further raises it to 88.3%. For the Amazon dataset, a test accuracy of 65.4% can be achieved by only using the structured data while this can be raised up to 68.4% by using the CNN features from the ResNet $conv5\_1$ layer (see Figure. 15). It is important to note that for Foods and Amazon datasets, different layers yielded the best accuracy and hence it emphasizes importance of exploring multiple layers in a CNN feature transfer workload.

Figure 16: Accuracy lifts obtained by incorporating HOG descriptors and CNN features for logistic regression model with $||\ell||_1$ regularization.



We also evaluate the effectiveness of using CNN features as image representations compared to traditional image feature extraction methods such as "Histogram of Oriented Gradients (HOG)" [33]. For the Foods and Amazon datasets, incorporating HOG descriptors with the structured features raises the accuracy from 85.2% to 86.5% and 65.4% to 66.3% respectively. As mentioned earlier, by using CNN features this can be further raised up to 88.3% and 68.4% for Foods and Amazon datasets respectively.

Table 4: Runtime breakdown for the images read time and 1^{st} iteration of the Logistic Regression for each feature layer when using different CNNs and **Foods** dataset with different number of nodes (Layer indices starts from the top and runtimes are in minutes).

		ResNet50/5L				AlexNet/4L				VGG16/3L			
		Number of nodes				Number of nodes				Number of nodes			
		1	2	4	8	1	2	4	8	1	2	4	8
Layer	5	19.0	9.5	4.5	2.3								
	4	3.8	1.8	0.9	0.4	3.7	2.1	1.2	0.7				
	3	2.7	1.3	0.7	0.4	2.4	1.3	0.7	0.5	43.0	22.0	11.0	5.4
	2	2.6	1.3	0.6	0.3	1.1	0.6	0.3	0.2	1.0	0.5	0.3	0.2
	1	1.8	0.9	0.4	0.2	0.3	0.2	0.1	0.1	0.3	0.2	0.1	0.1
	total	29.9	14.8	7.1	3.6	7.5	4.2	2.3	1.5	44.3	22.7	11.4	5.7
Read images		3.7	2.0	1.1	0.7	3.9	2.1	1.2	0.8	4.6	2.5	1.4	0.9

Figure 15: Accuracy lifts obtained by incorporating image features from different layers for logistic regression model with $||\ell||_1$ regularization.