Vista: Declarative Feature Transfer from Deep CNNs at Scale

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

Abstract

Scalable systems for machine learning (ML) are largely siloed into dataflow systems for structured data and deep learning systems for unstructured data. This gap has left workloads that jointly analyze both forms of data with poor systems support, leading to both low system efficiency and grunt work for users. We bridge this gap for an important class of such workloads: *feature transfer* from deep convolutional neural networks (CNNs) for analyzing images along with structured data. Executing feature transfer on scalable dataflow and deep learning systems today faces two key systems issues: *inefficiency* due to redundant computations and *crash-proneness* due to mismanaged memory. We present Vista, a new data system that resolves these issues by elevating this workload to a declarative level on top of dataflow and deep learning systems. Vista automatically optimizes the configuration and execution of this workload to reduce both computational redundancy and the potential for crashes. Experiments on real datasets show that apart from making feature transfer easier, Vista avoids crashes and reduces runtimes by 58% to 92% compared to baselines.

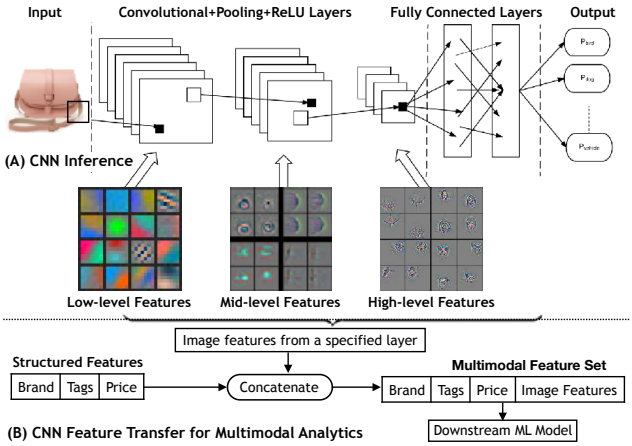

Figure 1. (A) Simplified illustration of a typical deep CNN and its hierarchy of learned feature layers(based on [57]). (B) Illustration of the CNN feature transfer workflow for multimodal analytics.

1 Introduction and Motivation

Deep CNNs achieve near-human accuracy for many image analysis tasks [29, 42]. Thus, there is growing interest in using CNNs to exploit images in analytics applications that have so far relied mainly on structured data. But ML systems today have a dichotomy: dataflow systems (e.g., Spark [56]) are popular for structured data [4, 45], while deep learning systems (e.g., TensorFlow [19]) are needed for CNNs. This dichotomy means the systems issues of workloads that combine both forms of data are surprisingly ill understood. In this paper, we present a new system that closes this gap for a popular form of such workloads: *feature transfer* from CNNs.

Example (Based on [44]). Consider a data scientist, Alice, at an online fashion retailer building a product recommender system (see Figure 1). She uses structured features (e.g., price, brand, user clicks, etc.) to build an interpretable ML model (e.g., logistic regression or decision tree) to predict product ratings. She then has a hunch that including product images can raise ML accuracy. So, she uses a pre-trained deep CNN (e.g., ResNet50 [31]) on the images to extract a *feature layer*: a vector representation of an image produced by the CNN. Deep CNNs produce a series of feature layers; each layer automatically captures a different level of abstraction from low-level patterns to high-level abstract shapes [30, 42], as Figure 1(A) illustrates. Alice concatenates her chosen feature layer with the structured features and trains her "downstream" model. Figure 1(B) illustrates this workflow. She then tries a few other feature layers instead to check if the downstream model's accuracy goes up.

Importance of Feature Transfer. Feature transfer is a form of "transfer learning" that mitigates two key pains of training deep CNNs from scratch [3, 16, 47]: the number of labeled images needed is lower, often by an order of magnitude [16, 55], and the time and resource costs of training are lower, even by two orders of magnitude [3, 16]. These benefits arise because the CNN's features help the downstream model analyze the image more easily. Overall, feature transfer is now popular in many domains, including recommender systems [44], visual search [36] (product images), healthcare (tissue images) [28], nutrition science (food images) [8], and computational advertising (ad images).

Bottleneck: Trying Multiple Layers. Recent work in ML showed that it is *critical to try multiple layers* for feature transfer because different layers yield different accuracies and it is impossible to tell upfront which layer will be best [16, 21, 27, 55]. But trying multiple layers becomes a bottleneck

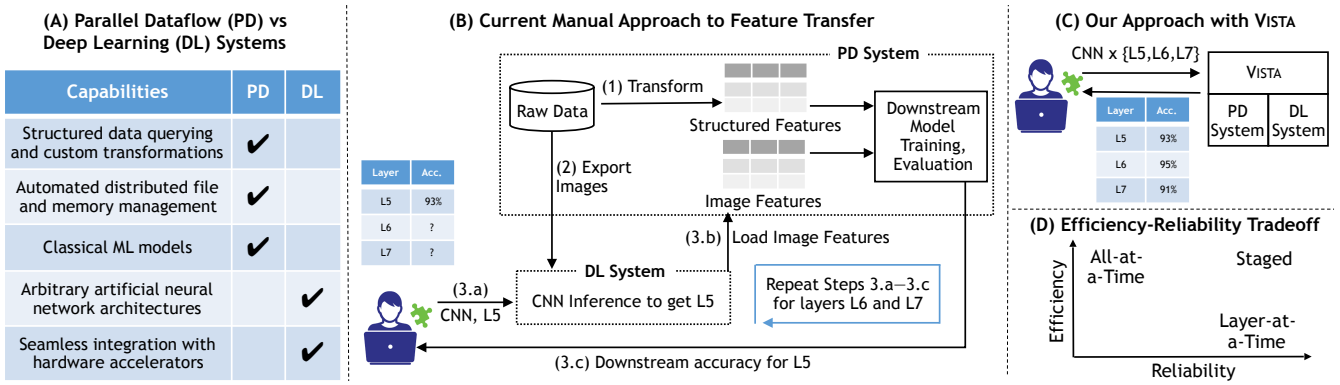

Figure 2. (A) Comparing the analytics-related capabilities of parallel dataflow (PD) systems and deep learning (DL) systems. (B) Current manual approach of executing feature transfer at scale straddling PD and DL systems. The steps in the manual workflow are numbered. Step 3 (a-b-c) is repeated for every feature layer of interest. (C) The "declarative" approach in Vista. (D) Tradeoffs of alternative execution plans on efficiency (runtimes) and reliability (crash-proneness).

for data scientists running large-scale ML on a cluster because it can slow down their analysis, e.g., from an hour to several hours (Section 5), and/or raise resource costs.

1.1 Current Approach and Systems Issues

The common approach to feature transfer at scale is a *tedious manual process* straddling deep learning (DL) systems and parallel dataflow (PD) systems. These systems present a dichotomy, as Figure 2(A) shows. PD systems support queries and manage distributed memory for structured data but do not support DL natively. DL systems support complex CNNs and hardware accelerators but need manual partitioning of files and memory for distributed execution. Moreover, data scientists often prefer interpretable ML models on structured features [6]; thus, a DL system alone is too limiting.

Figure 2(B) illustrates the manual process. Suppose Alice tries layers L5 to L7 (say) from a given CNN. She first runs CNN inference in TensorFlow to write out (*materialize*) L5 for all images in her dataset. She loads this large data file with image features into Spark, joins it with the structured data, and runs MLlib [45] for downstream training. She repeats all this for L6 and then for L7. Apart from being tedious grunt work, this process faces two key systems issues:

(1) Inefficiency. Extracting a higher layer (say, L6) requires a *superset of the inference computations* needed for a lower layer (say, L5). So, the manual process may have high *computational redundancy*, which wastes runtime.

(2) Crash-proneness. One might ask: *why not write out all layers in one go to save time?* Alas, CNN feature layers can be very large, e.g., one of ResNet50's layers is 784kB but the image is only 14kB [31]. So, 10GB of data blows up to 560GB for just one layer! Forcing ML users to handle such large intermediate data files on PD systems can easily cause system crashes due to memory mismanagement.

1.2 Our Proposed Approach

We resolve the above issues by elevating scalable feature transfer to a "declarative" level and automatically optimizing its execution. We want to retain the benefits of both PD and DL systems without reinventing their current capabilities (Figure 2(A)). Thus, we build a new data system middleware we call Vista *on top* of PD and DL systems, as Figure 2(C) illustrates. To make practical adoption easier, we believe it is crucial to *not modify the code* of the underlying PD and DL systems; this also lets us leverage future improvements to those systems. Vista is based on three design decisions: (1) *Declarativity* to simplify specification, (2) *Execution Optimization* to reduce runtimes, and (3) *Automated Memory and System Configuration* to avoid memory-related crashes.

(1) Declarativity. Vista lets users specify just *what* CNNs and layers to try, not *how* to run them. It invokes the DL system to run CNN inference, loads and joins image features with structured data, and runs downstream training on the PD system. Since Vista, not the user, handles how layers are materialized, it can optimize execution in non-trivial ways.

(2) Execution Optimization. We characterize the memory use behavior of this workload in depth, including key crash scenarios. This helps us bridge PD and DL systems, since PD systems do not understand CNNs and DL systems do not understand joins or caching. We compare alternative *execution plans* with different efficiency–reliability tradeoffs, as Figure 2(D) shows. The "Layer-at-a-Time" plan simply automates the manual process. It is reliable due to its low memory footprint, but it has high computational redundancy. At the other end, "All-at-a-Time" materializes all layers of interest in one go (Section 4). It avoids redundancy but is prone to memory-related crashes. We then present a new plan used in Vista that offers the best of both worlds: "Staged" execution; it *interleaves* the DL and PD systems' operations by enabling *partial CNN inference*.

(3) Automated Memory and System Configuration. Finally, we explain how key system tuning knobs affect this workload: apportioning memory for caching data, CNNs, and feature layers; data partitioning; and physical join operator. Using our insights, we build an *end-to-end automated optimizer* in Vista to configure both the PD and DL systems to run this workload efficiently and reliably.

Implementation and Evaluation. We prototype Vista on top of two PD systems, Spark and Ignite [22], with TensorFlow as the DL system. We chose these systems due to their popularity but note that our ideas are general and applicable to other DL systems (e.g., PyTorch [10]) and PD systems (e.g., Greenplum [9]) as well. Our API is in Python. We perform an extensive empirical evaluation of Vista using 2 real-world multimodal datasets and 3 deep CNNs. Vista avoids many crash scenarios and reduces total runtimes by 58% to 92% compared to existing baseline approaches.

Relationship to Database Optimization. Our approach is inspired by query optimization in the database literature [49]. But our execution plans and optimizer have no counterparts in databases because they treat CNNs as black-box user-defined functions that they do not rewrite. In contrast, Vista treats CNNs as first-class operations, understands their memory footprints, rewrites their inference, and optimizes this workload in a principled and holistic manner.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first work on the systems principles of integrating PD and DL systems to optimize scalable feature transfer from CNNs.
- We characterize the memory use behavior of this workload in depth, explain the efficiency–reliability tradeoffs of alternative execution plans, and present a new CNN-aware optimized execution plan.
- We create an automated optimizer to configure the system and optimize its execution to offer both high efficiency and high reliability.
- We prototype our ideas to build Vista on top of a PD and DL system. We compare Vista against baseline approaches using multiple real-world datasets and deep CNNs. Unlike the baselines, Vista never crashes and is also faster by 58% to 92%.

2 Background

We provide some background from the ML and data systems literatures to help understand the rest of this paper. We defer discussion of other related work to Section 6.

Deep CNNs. CNNs are a type of neural networks specialized for images [30, 42]. They learn a hierarchy of parametric features using layers of various types (see Figure 1(A)): *convolutions* learn filters to extract features; *pooling* subsamples features; *non-linearity* applies a non-linear function (e.g., ReLU) to all features; and *fully connected* is a set of perceptrons. All parameters are trained using backpropagation [41]. CNNs typically surpass older hand-crafted image features such as SIFT and HOG in accuracy [25, 43]. Training a CNN *from scratch* incurs massive costs: they typically need many GPUs for reasonable runtimes [3], huge labeled datasets, and complex hyper-parameter tuning [30].

Transfer Learning with CNNs. Transfer learning mitigates the cost and labeled data requirements of training deep CNNs from scratch [47]. When transferring CNN features, no single layer is universally best for accuracy; the "more similar" the target task is to the source task (e.g., ImageNet classification), the better the higher layers will likely be [16, 21, 27, 55]. Also, lower layer features are often much larger; so, simple feature selection such as extra pooling is typically used [21]. Such feature transfer underpins recent breakthrough applications of CNNs in detecting cancer [28], diabetic retinopathy [52], facial analysis [54], and multimodal recommendation systems [44].

Spark, Ignite, and TensorFlow. Spark and Ignite are popular distributed memory-oriented data systems [2, 22, 56]. At their core, both use a distributed collection of key-value pairs as the data abstraction. They support many dataflow operations, including relational operations and MapReduce. Spark's collection, called a Resilient Distributed Dataset or RDD, is immutable, while Ignite's is mutable. Spark holds data in memory and supports disk spills; Ignite uses memory as a cache for data on disk. Both systems support user-defined functions (UDFs) to let users run ML algorithms directly on large datasets, e.g., with Spark MLlib [45].

TensorFlow (TF) is a system for training and running neural networks [18, 19]. Models in TF are specified as a "computational graph," with nodes representing operations over "tensors" (multi-dimensional arrays) and edges representing dataflow. *TensorFrames* and *SparkDL* are libraries that integrate Spark and TF [14, 15]. *TensorFrames* lets users process Spark data tables using TF code, while *SparkDL* offers pipelines to integrate neural networks into Spark queries and distribute hyper-parameter tuning. *SparkDL* is the most closely related work to ours, since it too supports transfer learning. But unlike Vista, *SparkDL* does not support trying multiple layers of a CNN nor does it optimize this workload's execution. Thus, both the functionality and techniques of Vista are complementary to *SparkDL*.

3 Preliminaries and Overview

We now formally describe our problem setting, explain our assumptions, and present an overview of Vista.

3.1 Definitions and Data Model

We start by defining some terms and notation to formalize the data model of *partial CNN inference*. We will use these terms in the rest of this paper.

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

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 that will be used to explain our techniques.

Definition 3.2. *A* TensorList *is an indexed list of tensors of potentially different shapes.*

Definition 3.3. *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.*

Definition 3.4. *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.[1] *We use* $\hat{f}_i$ *to denote* $f_i(\ldots f_2(f_1(\cdot))\ldots)$.

Definition 3.5. CNN inference. *Given a CNN* f *and a shape-compatible image tensor* t, CNN inference *is the process of computing* $f(t)$.

Definition 3.6. 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* $\hat{f}_{i \rightarrow j}$.

All major CNN layers–convolutional, pooling, non-linearity, and fully connected–are just TensorOps. The above definitions capture a crucial aspect of partial CNN inference: data flowing through the layers produces a sequence of tensors.

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; we can state it more precisely as follows:

[1]We use sequential (chain) CNNs for simplicity of exposition; it is easy to extend our definitions to DAG-structured CNNs such as DenseNet [35].

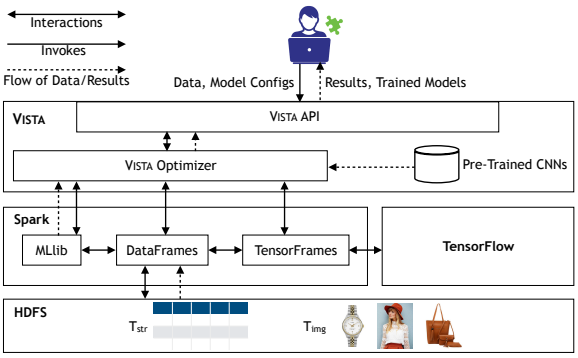

Figure 3. System architecture of the VISTA prototype on top of the Spark-TensorFlow combine. The prototype on Ignite-TenforFlow is similar and skipped for brevity.

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

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

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

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

Step (2) runs partial CNN inference to materialize layer l and flattens it by vectorizing (vec). Step (3) concatenates structured and image features using a key-key join. Step (4) trains M on the concatenated feature vector. Pooling can be inserted before vec to reduce dimensionality for M [21]. The current approach (Figure 2(B)) runs the above queries as such, i.e., materialize layers *manually* and *independently* as flat files and transfer them; we call this execution plan *Layer-at-a-Time*. This plan is cumbersome, inefficient due to *redundant* partial CNN inference, and/or is prone to crashes due to inadvertently mismanaged memory.

We make a few simplifying assumptions for tractability in this first paper on this problem. First, we assume that f is from a roster of well-known CNNs. We currently support AlexNet [39], VGG16 [50], and ResNet50 [31] due to their popularity in real feature transfer applications [44, 54]. We leave support for arbitrary CNNs to future work. Second, we support only one image per example; we leave support for multiple images per example to future work. Third, we use linear classifiers in MLlib for M; this choice is orthogonal to this paper's focus but it lets us study our workload's tradeoffs in depth. Finally, we assume enough secondary storage is available for disk spills and optimize the use of distributed memory; this is a standard assumption in PD systems.

3.3 System Architecture and API

We prototype VISTA as a library on top of Spark-TF and Ignite-TF environments. Due to space constraints, we explain the architecture of only the Spark-TF prototype; the Ignite-TF one is similar. VISTA has three components, as Figure 3 illustrates: (1) a "declarative" API, (2) a roster of popular named deep CNNs with numbered feature layers,

```
/**
 * * * * * * * * * * * * * * * * * * Input Parameters * * * * * ** * * * * *
 * name          : Name given to the experiment
 * mem_sys       : System memory available on a worker node
 * n_nodes       : # of nodes in the cluster
 * cpu_sys       : # of CPUs available on a worker node
 * model         : CNN model name. Possible values
 * n_layers      : # of layers from the last layer of the CNN to be explored
 * start_layer   : Starting layer of the ConvNet.
 * ml_func       : Function pointer which implements the downstream ML model
 * struct_input  : Input path for the strucutred input
 * images_input  : Input path for the images
 * n             : # of records
 * dS            : # of structured features
**/
vista = Vista("vista-example", 32, 8, 8, 'alexnet', 4, 0, ml_func,
            'hdfs://../foods.csv', 'hdfs://.../foods-images', 20129, 130)


//Initiate CNN feature transfer workload. Function returns the training
//accuracies for each evaluated layer
train_accuracies = vista.run()
```

Figure 4. Vista API and sample usage showing values for the input parameters and invocation.

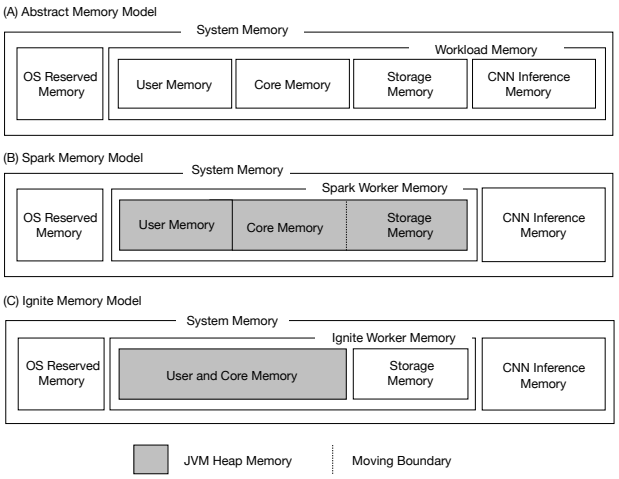

Figure 5. (A) Our abstract model of distributed memory apportioning. (B,C) How our model maps to Spark and Ignite.

and (3) the Vista optimizer. Our Python API (see Figure 4) expects 4 major groups of inputs. First is the system environment (memory, number of cores, and number of nodes). Second, a deep CNN f and the number of feature layers $|L|$ (starting from the top most layer) to explore. Third, the downstream ML routine M that handles the downstream model's evaluation and artifacts. Fourth, data tables T_{str} and T_{img} and statistics about the data. Our API returns the model evaluation output for each layer.

Under the covers, Vista invokes its optimizer (Section 4.3) to pick a fast and reliable set of choices for the logical execution plan (Section 4.2.1), system configuration parameters (Section 4.2.2), and physical execution decisions (Section 4.2.3). After configuring Spark accordingly, Vista runs within the Spark Driver process to control the execution. Vista injects UDFs to run (partial) CNN inference, i.e., f, $\hat{f}_l$, vec, and $\hat{f}_{i \to j}$ for the CNNs in its roster (currently, AlexNet, VGG16, and ResNet50). These UDFs specify the computational graphs for TF and invoke Spark's *DataFrames* and *TensorFrames* APIs with appropriate inputs based on our optimizer's decisions. Image and feature tensors are stored with our custom *TensorList* datatype. Finally, Vista invokes MLlib for downstream training on the concatenated feature vector and obtains $|L|$ trained downstream models. *Overall, Vista frees ML users from manually rewriting TF code, saving features as files, performing joins, or tuning Spark for running this workload at scale.*

4 Tradeoffs and Optimizer

We now characterize the abstract memory usage behavior of our workload in depth. We then map our memory model to Spark and Ignite. Finally, we use these insights to explain three dimensions of efficiency-reliability tradeoffs and apply our analyses to design the Vista optimizer.

4.1 Memory Use Characterization of Workload

It is important to understand and optimize the memory use behavior of the feature transfer workload, since mismanaged memory can cause frustrating system crashes and/or excessive disk spills or cache misses that raise runtimes. Apportioning and managing distributed memory carefully is a central concern for modern distributed data processing systems. Since our work is not tied to any specific dataflow system, we create an *abstract model of distributed memory apportioning* to help us explain the tradeoffs in a generic manner. These tradeoffs involve apportioning memory between intermediate data, CNN models, and working memory for UDFs. Such tradeoffs affect both reliability (avoiding crashes) and efficiency. We then highlight interesting new properties of our workload that can cause unexpected crashes or inefficiency, if not handled carefully.

Abstract Memory Model. In distributed memory-based dataflow systems, a worker's System Memory is split into two main regions: Reserved Memory for OS and other processes and Workload Memory, which in turn is split into Execution Memory and Storage Memory. Figure 5(A) illustrates the regions. Execution Memory is further split into User Memory and Core Memory; for typical relational/SQL workloads, the former is used for UDF execution, while the latter is used for query processing. Best practice guidelines recommend allotting most of System Memory to Storage Memory, while having enough Execution Memory to reduce disk spills or cache misses [5, 12, 13]. OS Reserved Memory is typically a few GBs. For our workload, however, we need to rethink these memory apportioning guidelines due to three new issues caused by CNNs and partial CNN inference that do not arise in traditional SQL workloads.

(1) The guideline of using most of System Memory for Storage and Execution no longer holds. In both Spark and Ignite, CNN inference in TF uses System Memory *outside* Storage and Execution regions. The memory footprint of

deep CNNs is non-trivial, e.g., AlexNet needs 2GB. For parallel query execution in PD systems, each execution thread will spawn its own CNN replica, multiplying the footprint.

(2) Many temporary objects are created when reading serialized CNNs to initialize TF, for buffers to read inputs, and to hold CNN features created by inference. All of these go under User Memory. The sizes of these objects depend on the number of examples in a data partition, the CNN, and L. These sizes could vary dramatically and also be very high, e.g., layer *fc6* of AlexNet has 4096 features but *conv5* of ResNet has over 400,000 features! Such complex memory footprint calculations will be too tedious for ML users.

(3) The downstream ML routine also copies features produced by TF into its own representations. Thus, Storage Memory should accommodate such intermediate data copies. Finally, Core Memory must accommodate the temporary objects created for processing the join.

Mapping to Spark's Memory Model. Spark allocates User, Core, and Storage Memory regions of our abstract memory model from the JVM Heap Space. With default configurations, Spark allocates 40% of the Heap Memory to User Memory region. The rest of the 60% is shared between the Storage and Core Memory regions. The Storage Memory–Core Memory boundary in Spark is not static. If needed, Core Memory automatically borrows from the Storage Memory evicting data partitions to the disk. Conversely, if Spark needs to load more data to memory, it borrows from Execution Memory. But there is a maximum threshold fraction of Storage Memory (default 50%) that is immune to eviction. Worker threads in Spark run in isolation with no access to shared memory.

Mapping to Ignite's Memory Model. Ignite treats both User and Core Memory regions as a single unified memory region and allocates the entire JVM Heap for it. This region is used to store the in-memory objects generated by Ignite during query processing and UDF execution. Storage Memory region of Ignite is allocated *outside* of JVM heap in the JVM native memory space. Unlike Spark, Ignite's in-memory Storage Memory region has a static size and uses an LRU cache for data stored on persistent storage. Unlike Spark, worker threads in Ignite can have access to shared memory (we exploit this in Vista, as explained later).

Memory-related Crash and Inefficiency Scenarios. The three twists explained above give rise to various unexpected system crash scenarios due to memory errors, as well as system inefficiencies. Manually handling them could frustrate data scientists and impede their ML exploration.

(1) CNN blowups. Serialized file formats of CNNs often underestimate their in-memory footprints. Along with the replication by multiple threads, CNN Inference Memory can be easily exhausted. If such blowups are not accounted when configuring the data processing system, and if they exceed available memory, the OS will kill the application.

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

(3) Very large data partitions. If a data partition is too big, the data processing system needs a lot of User and Core Execution Memory for query execution operations (e.g., for the join in our workload and *MapPartition*-style UDFs in Spark). If Execution Memory consumption exceeds the allocated maximum, it will cause the system to crash with out-of-memory error.

(4) Insufficient memory for Driver Program. All distributed data processing systems require a Driver program that orchestrates the job among workers. In our case, the Driver reads and creates a serialized version of the CNN and broadcasts it to the workers. To run the downstream ML task, the Driver has to collect partial results from workers (e.g., for *collect()* and *collectAsMap()* in Spark). Without enough memory for these operations, the Driver will crash.

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

4.2 Three Dimensions of Tradeoffs

The dimensions we discuss are largely orthogonal to each other but they affect reliability and efficiency collectively.

4.2.1 Logical Execution Plan Tradeoffs

Figure 6(A) illustrates the the *Layer-at-a-Time* plan (Section 3.2). As mentioned earlier, it has high computational redundancy; to see why, consider a popular deep CNN AlexNet with the last two layers *fc7 and fc8* used for feature transfer ($L = \{fc7, fc8\}$). This plan performs partial CNN inference for *fc7* (721 MFLOPS) independently of *fc8* (725 MFLOPS), incurring 99% redundant computations for *fc8*. An 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 size of raw images in a compressed format such as JPEG. Thus, if the join is pulled below inference, as shown in Figure 6(B), the shuffle costs of the join will go down. We call this slightly modified plan *Layer-at-a-Time-Reordered*. But this plan still has computational redundancy. The only way to remove redundancy is to break the independence of the $|L|$ queries and fuse them.

Consider the *All-at-a-Time* plan shown in Figure 6(C). It materializes all feature layers of L *in one go*, which avoids redundancy because CNN inference is not repeated. Features are stored as a *TensorList* in an intermediate table and joined with T_{str}. M is then trained on each feature layer

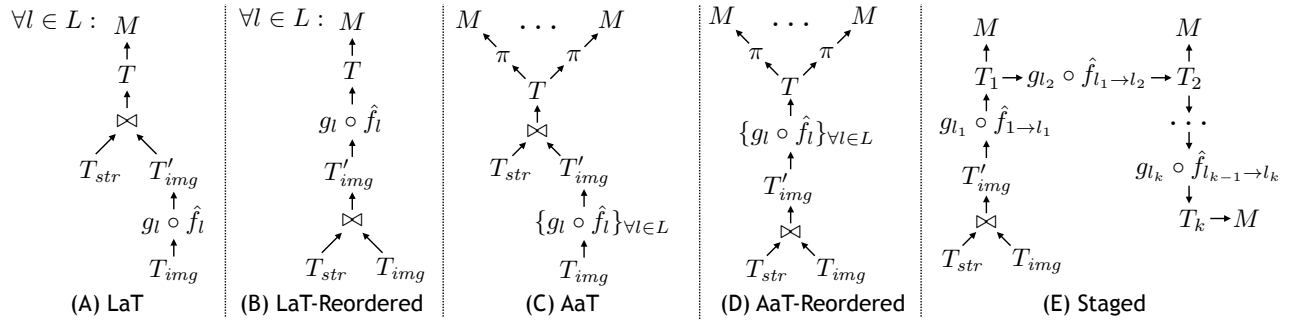

Figure 6. Alternative logical execution plans (let $k = |L|$). (A) Layer-at-a-Time (LaT), the de facto current approach. (B) Reordering the join operator in LaT. (C) All-at-a-Time (AaT) execution plan. (D) Reordering the join operator in AaT. (E) Our new Staged execution plan.

(concatenated with X) projected from the *TensorList*. *All-at-a-Time-Reordered*, shown in Figure 6(D), is a variant with the join pulled down. Alas, both of these plans have high memory footprints, since they materialize all of L at once. Depending on the memory apportioning (Section 4.1), this could cause system crashes or a lot of disk spills, which in turn raises runtimes.

To resolve the above issues, we create a logically new execution plan we call *Staged* execution, shown in Figure 6(E). It splits partial CNN inference across the layers in L and invokes M on branches off of the inference path; so, it stages out the materialization of the feature tensors. *Staged* offers the best of both worlds: it avoids computational redundancy, and it is reliable due to its lower memory footprints. Empirically, we find that *All-at-a-Time* and *All-at-a-Time-Reordered* are seldom much faster than *Staged* due to a peculiarity of deep CNNs. The former can be faster only if a CNN "quickly" (i.e., within a few layers and low FLOPs) converts the image to small feature tensors. But such a CNN architecture is unlikely to yield high accuracy, since it loses too much information too soon [30]. Indeed, no popular deep CNN has such an architecture. Thus, Vista only uses our new *Staged* execution plan (validated in Section 5).

4.2.2 System Configuration Tradeoffs

Logical execution plans are generic and independent of the PD system used. But as explained in Section 4.1, three key system configuration parameters matter for reliability and efficiency: degree of parallelism in a worker, data partition sizes, and memory apportioning. Once again, while the tuning of such parameters is well understood for SQL and MapReduce workloads [34, 53], we need to rethink them due to the properties of CNNs and partial CNN inference.

Naively, one might choose the following settings that may work well for SQL workloads: the degree of parallelism is the number of cores on a node; allocate few GBs for User and Core Execution Memory; use most of the rest of memory for Storage Memory; use the default number of partitions in the PD system. But for the feature transfer workload, these settings can cause crashes or inefficiencies.

For example, a higher degree of parallelism increases the worker's throughput but also raises the CNN models' footprint, which in turn requires reducing Execution and Storage Memory. Reducing Storage Memory can cause more disk spills, especially for feature layers, and raise runtimes. Worse still, User Memory might also become too low, which can cause crashes during CNN inference. Lowering the degree of parallelism reduces the CNN models' footprint and allows Execution and Storage Memory to be higher, but too low a degree of parallelism means workers will get underutilized.[2] This in turn can raise runtimes, especially for the join and the downstream training. Finally, too low a number of data partitions can cause crashes, while too high a value leads to high overheads. Overall, we see multiple non-trivial systems tradeoffs that are tied to the CNN and its feature layer sizes. It is unreasonable to expect ML users to handle such tradeoffs manually. Thus, Vista automates these decisions in a feature transfer-aware manner.

4.2.3 Physical Execution Tradeoffs

Physical execution decisions are closer to the specifics of the underlying PD system. We discuss the tradeoffs of two such decisions that are common in PD systems and then explain what Spark and Ignite specifically offer.

First is the physical join operator used. The two main options for distributed joins are *shuffle-hash* and *broadcast*. In shuffle-hash join, base tables are hashed on the join attribute and partitioned into "shuffle blocks." Each shuffle block is then sent to an assigned worker over the network, with each worker producing a partition of the output table using a local sort-merge join or hash join. In broadcast join, each worker gets a copy of the smaller table on which it builds a local hash table before joining it with the outer table without any shuffles. If the smaller table fits in memory, broadcast join is typically faster due to lower network and disk I/O costs.

[2] We note, however, that in our current prototypes, every TF invocation by a worker uses all cores on the node regardless of how many cores are assigned to that worker. Nevertheless, one TF invocation per used core helps increase overall throughput and reduce runtimes.

Second is the persistence format for in-memory storage of intermediate data. Since feature tensors can be much larger than raw images, this decision helps avoid/reduce disk spills or cache misses. The two main options are *deserialized* format or *compressed serialized* format. While the serialized format can reduce memory footprint and thus, reduce disk spills/cache misses, it incurs additional computational overhead for translating between formats. To identify potential disk spills/cache misses and determine which format to use, we estimate the size of intermediate data tables $|T_i|$ (for $i \in L$). Vista can automatically estimate $|T_i|$ because it knows the sizes of the feature tensors in its CNN roster and understands the internal record format of the PD system. For the interested reader, more details are available in Appendix A of our addendum [17].

Spark supports both shuffle-hash join and broadcast join implementations, as well as both deserialized and compressed serialized in-memory storage formats. In Ignite, data is shuffled to the corresponding worker node based on the partitioning attribute during data loading itself. Thus, a key-key join can be performed using a local hash join without any additional data shuffles, if we use the same data partitioning function for both tables. Ignite always stores intermediate in-memory data in a compressed binary format.

4.3 The Optimizer

We now explain how the Vista optimizer navigates all the tradeoffs in a holistic and automated way to improve both reliability and efficiency. Table 1 lists the notation used.

Optimizer Formalization and Simplification. Table 1(A) lists the inputs given by the user. From these inputs, Vista infers the sizes of the structured data table ($|T_{str}|$), the images table ($|T_{img}|$), and all intermediate data tables ($|T_i|$ for $i \in L$) shown in Figure 6(E). Vista also looks up the CNN's serialized size $|f|_{ser}$, runtime memory footprint $|f|_{mem}$, and runtime GPU memory footprint $|f|_{mem\_gpu}$ from its roster, in which we store these statistics. Then, Vista calculates the runtime memory footprint of the downstream model $|M|$ based on the specified M and the largest total number of features (based on L). For instance, for logistic regression, $|M|$ is proportional to $(|X| + \max_{l \in L} |g_l(\hat{f}_l(I))|)$. Table 1(B) lists the variables whose values are set by the optimizer. We define two quantities that capture peak intermediate data sizes to help our optimizer set memory variables reliably:

$$s_{single} = \max_{1 \le i \le |L|} |T_i| \tag{5}$$

$$s_{double} = \max_{1 \le i \le |L|-1} (|T_i| + |T_{i+1}|) - |T_{str}| \tag{6}$$

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* and $mem_{storage}$. Raising *cpu* increases parallelism, which could reduce runtimes.

Table 1. Notation for Section 4 and Algorithm 1.

Symbol	Description
(A) Inputs given by user to Vista	
T_{str}	Structured features table
T_{img}	Images table
f	CNN model in our roster
L	Set of feature layer indices of f to transfer
M	Downstream ML routine
n_{nodes}	Number of worker nodes in cluster
mem_{sys}	Total system memory available in a worker node
mem_{GPU}	GPU memory if GPUs are available
cpu_{sys}	Number of cores available in a worker node
(B) System variables/decisions set by Vista Optimizer	
$mem_{storage}$	Size of Storage Memory
mem_{user}	Size of User Memory
cpu	Number of cores assigned to a worker
n_p	Number of data partitions
$join$	Physical join implementation (*shuffle* or *broadcast*)
$pers$	Persistence format (*serialized* or *deserailized*)
(C) Other fixed (but adjustable) system parameters	
mem_{os_rsv}	Operating System Reserved Memory (default: 3 GB)
mem_{core}	Core Memory as per system specific best practice guidelines (e.g. Spark default: 2.4 GB)
p_{max}	Maximum size of data partition (default: 100 MB)
b_{max}	Maximum broadcast size (default: 100 MB)
cpu_{max}	Cap recommended for *cpu* (default: 8)
α	Fudge factor for size blowup of binary feature vectors as JVM objects (default: 2)

But it also raises the CNN inference memory needed for TF, which forces $mem_{storage}$ to be reduced, thus increasing potential disk spills/cache misses for T_i's and raising runtimes. This tension is captured by the following objective function:

$$\min_{cpu, n_p, mem_{storage}} \frac{\tau + \max(0, \frac{s_{double}}{n_{nodes}} - mem_{storage})}{cpu} \tag{7}$$

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 "constant" for this optimization. The second term captures disk spill costs for T_i's. The denominator captures the degree of parallelism. While this objective is ideal, it is largely impractical and needlessly complicated for our purposes due to three reasons. (1) Estimating τ is tedious, since it involves join costs, data loading costs, downstream model costs, etc. (2) More importantly, we hit a point of diminishing returns with *cpu* quickly, since CNN inference typically dominates total runtime and TF anyway uses all cores regardless of *cpu*. That is, this workload's speedup against *cpu* will be quite sub-linear

(confirmed by Figure 10(C) in Section 5). Empirically, we find that about 7 cores typically suffice; interestingly, a similar observation is made in Spark guidelines for purely relational workloads [12, 14]. Thus, we cap *cpu* at $cpu_{max} = 8$. (3) Given the cap on *cpu*, we can just drop the term minimizing disk spill/cache miss costs, since s_{double} will typically be smaller than the total memory (even after accounting for the CNNs) due to the above cap.

Overall, our insights above yield a simpler objective that is still a reasonable surrogate for minimizing runtimes:

$$\max_{cpu,\, n_p,\, mem_{storage}} cpu \tag{8}$$

The constraints for the optimization are as follows:

$$1 \leq cpu \leq \min\{cpu_{sys}, cpu_{max}\} - 1 \tag{9}$$

$$mem_{user} = \begin{cases} \text{(a) no shared memory:} \\ \quad cpu \times \max\{|f|_{ser} + \alpha \times \lceil s_{single}/n_p \rceil, |M|\}, \\ \text{(b) shared memory:} \\ \quad \max\{|f|_{ser} + cpu \times \alpha \times \lceil s_{single}/n_p \rceil, \\ \quad\quad cpu \times |M|\} \end{cases} \tag{10}$$

$$mem_{os\_rsv} + cpu \times |f|_{mem} + mem_{user} + mem_{core} + mem_{storage} < mem_{sys} \tag{11}$$

$$n_p = z \times cpu \times n_{nodes}, \text{ for some } z \in \mathbb{Z}^+ \tag{12}$$

$$\lceil s_{single}/n_p \rceil < p_{max} \tag{13}$$

If GPUs are available:

$$cpu \times |f|_{mem\_gpu} < mem_{GPU} \tag{14}$$

Equation 9 caps *cpu* and leaves a core for the OS. Equation 10 captures User Memory for reading CNN models and invoking TF, copying materialized feature layers from TF, and holding M. If worker threads have access to shared memory, the serialized CNN model need not be replicated, as Equation 10(b) shows. $cpu \times |f|_{mem}$ is the CNN Inference Memory needed for TF. Equation 11 constrains the total memory as per Figure 5. If there are GPUs, total GPU memory footprint $cpu \times |f|_{mem\_gpu}$ should be bounded by available GPU memory mem_{GPU} as per Equation 14. Equation 12 requires n_p to be a multiple of the number of worker processes to avoid skews, while Equation 13 bounds the size of an intermediate data partition as per system specific guidelines [1].

Optimizer Algorithm. Given our above observations, the algorithm is simple: linear search on *cpu* to satisfy all constraints.[3] Algorithm 1 presents it formally. If the **for** loop completes without returning, there is no feasible solution, i.e., System Memory is too small to satisfy some constraints, say, Equation 11. In this case, Vista notifies the user, and the user can provision machines with more memory. Otherwise, we have the optimal solution. The other variables are set based on the constraints. We set *join* to *broadcast* if the predefined maximum broadcast data size constraint is satisfied; otherwise, we set it to *shuffle*. Finally, as per Section 4.2.3, *pers* is set to *serialized*, if disk spills/cache misses are likely (based on the newly set $mem_{storage}$). 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.

[3]We explain our algorithm for the CPU-only scenario with no shared memory among workers. It is straightforward to extend to the other settings.

Algorithm 1 The Vista Optimizer Algorithm.

procedure OptimizeFeatureTransfer:
  inputs: see Table 1(A)
  outputs: see Table 1(B)
  for $x = \min\{cpu_{sys}, cpu_{max}\} - 1$ **to** 1 **do** ▷ Linear search
    $n_p \leftarrow$ NumPartitions$(s_{single}, x, n_{nodes})$
    $mem_{worker} \leftarrow mem_{sys} - mem_{os\_rsv} - x \times |f|_{mem}$
    $mem_{user} \leftarrow x \times \max\{|f|_{ser} + \alpha \times \lceil s_{single}/n'_p \rceil, |M|\}$
    if $mem_{worker} - mem_{user} > mem_{core}$ **then**
      $cpu \leftarrow x$
      $mem_{storage} \leftarrow mem_{worker} - mem_{user} - mem_{core}$
      $join \leftarrow shuffle$
      if $|T_{str}| < b_{max}$ **then**
        $join \leftarrow broadcast$
      $pers \leftarrow deserialized$
      if $mem_{storage} < s_{double}$ **then**
        $pers \leftarrow serialized$
      return $(mem_{storage}, mem_{user}, cpu, n_p, join, pers)$
  throw Exception(No feasible solution)

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

5 Experimental Evaluation

We empirically validate if Vista is able to improve efficiency and reliability of feature transfer workloads. We then drill into how it navigates the tradeoff space.

Datasets. We use two real-world public datasets: *Foods* [8] and *Amazon* [32]. *Foods* has about 20,000 examples with 130 structured numeric features such as nutrition facts along with their feature interactions and an image of each food item. The target represents if the food is plant-based or not. *Amazon* is larger, with about 200,000 examples with structured features such as price, title, and categories, as well as a product image. The target is the sales rank, which we binarize as a popular product or not. We pre-processed title strings to get 100 numeric features (an "embedding") using Doc2Vec [40]. We convert the indicator vector of categories

to 100 numeric features using PCA. All images are resized to 227×227 resolution, as needed by popular CNNs. Overall, *Foods* is about 300 MB in size; *Amazon* is 3 GB. While these can fit on a single node, multi-node parallelism helps reduce completion times for long running ML workloads; also note that intermediate data sizes during feature transfer can be even 50x larger. We will release all of our data pre-processing scripts and system code on our project webpage.

Workloads. We use three ImageNet-trained deep CNNs: AlexNet [39], VGG16 [50], and ResNet50 [31], obtained from [7]. They complement each other in terms of model size [23]. We select the following layers for feature transfer from each: *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 [31]). Following standard practices [16, 55], we apply max pooling on 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 for research [48]. Each node has 32 GB RAM, Intel Xeon @ 2.00GHz CPU with 8 cores, and 300 GB Seagate Constellation ST91000640NS HDDs. All nodes run Ubuntu 16.04. We use Spark v2.2.0 with *TensorFrames* v0.2.9, TensorFlow v1.3.0, and Ignite v2.3.0. 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. Ignite is configured with native persistence enabled; each node runs one worker. All runtimes reported are the average of three runs with 90% confidence intervals.

5.1 End-to-End Reliability and Efficiency

We compare Vista with five baselines: three naive and two strong. *Layer-at-a-Time-1* (1 CPU per Executor), *Layer-at-a-Time-5* (5 CPUs), and *Layer-at-a-Time-7* (7 CPUs) capture the current dominant practice of Layer-at-a-Time execution (Section 3.2). Spark is configured based on best practices [5, 12] (29 GB JVM heap, shuffle join, deserialized, and defaults for all other parameters, including n_p and memory apportioning). Ignite is configured with a 4 GB JVM heap, 25 GB off-heap Storage Memory, and n_p set to the default 1024. *Layer-at-a-Time-5 with Pre-mat* and *All-at-a-Time* are strong baselines based on our tradeoff analyses in Section 4.2.1. In *Layer-at-a-Time-5 with Pre-mat*, the lowest layer specified (e.g., *conv5* for AlexNet) is materialized beforehand and used instead of raw images for all subsequent CNN inference; *Pre-mat* is time spent on pre-materializing the lowest layer specified. *All-at-a-Time* is an alternative plan explained in Section 4.2.1; we use 5 CPUs per Executor. For *Layer-at-a-Time-5 with Pre-mat* and *All-at-a-Time*, we explicitly apportion CNN Inference memory. Note that *Layer-at-a-Time-5 with Pre-mat* and *All-at-a-Time* actually need parts of our code from Vista. Figure 7 presents the results. For brevity sake, we abbreviate Layer-at-a-Time to LaT and All-at-a-Time to AaT.

Overall, Vista improves reliability and/or efficiency across the board. On Spark-TF, *LaT-5* and *LaT-7* crash on both datasets for VGG16; *All-at-a-Time* crashes on *Amazon* for VGG16 and ResNet50. On Ignite-TF, *LaT-7* crashes for all CNNs on *Amazon*, while for ResNet50, *LaT-7* on *Foods* and *AaT* on *Amazon* also crash due to memory-related issues (Section 4.1). When *AaT* does not crash, its efficiency is comparable to Vista, which validates our analysis in Section 4.2.1. *LaT-5 with Pre-mat* does not crash, but its runtimes are comparable to *LaT-5* and mostly higher than Vista. This is because the layers of AlexNet and ResNet are much larger than the images, which raises data I/O and join costs. One might wonder if more careful tuning could avoid the crashes with *AaT* and *LaT*. But that forces ML users to waste time wrestling with low-level systems tradeoffs (Section 4)–time they can now spend on further ML analysis instead.

Compared to *LaT-7*, Vista is 62%–72% faster; compared to *LaT-1*, 58%–92%. These gains arise mainly because Vista removes redundancy in partial CNN inference.Of course, the exact gains depend on the CNN and L: if more of the higher layers are tried, the more redundancy there is and the faster Vista will be. We also ran this experiment on GPUs; the trends were the same although all runtimes went down. Due to space constraints, we provide the detailed runtime breakdowns and the GPU results in our addendum for interested readers [17]. Overall, Vista never crashes and offers the best (or near-best) efficiency.

Accuracy. All approaches in Figure 7 (including Vista) yield identical downstream models (and thus, same accuracy) for a given CNN layer. We saw test F1 score lifts of 3% to 5% for the downstream model with feature transfer. As expected, the lift varies across CNNs and layers. For instance, on *Foods*, structured features alone give 80.2% accuracy. Adding ResNet50's *conv-5-3* layer raises it to 85.4%, a large lift in ML terms. But using the last layer *fc-6* gives only 83.5%. *Amazon* exhibited similar trends. We provide more details in Appendix D of our addendum for interested readers [17].

5.2 Drill-Down Analysis of Vista's Tradeoffs

We now analyze how Vista navigates the tradeoffs explained in Section 4. We use Vista on Spark-TF, since it is faster than Ignite-TF. We use the less resource-intensive *Foods* dataset but alter it semi-synthetically for some experiments to study Vista runtimes in new operating regimes. In particular, when specified, vary the data scale by replicating records (say, "4X") or varying the number of structured features (with random values). For uniformity sake, unless specified otherwise, we use all 8 workers, fix *cpu* to 4, and fix Core Memory to 60%

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

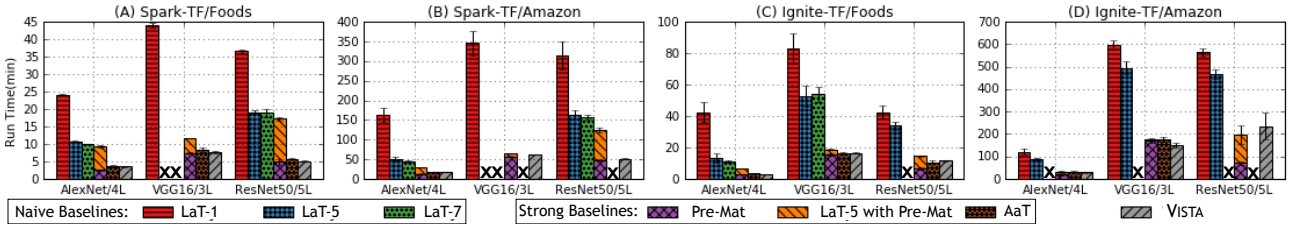

Figure 7. End-to-end reliability and efficiency. LaT stands for Layer-at-a-Time; AaT stands for All-at-a-Time. "×" means the system crashed. Overall, Vista offers the best or near-best runtimes and never crashes, while the alternatives are much slower or crash in some cases.

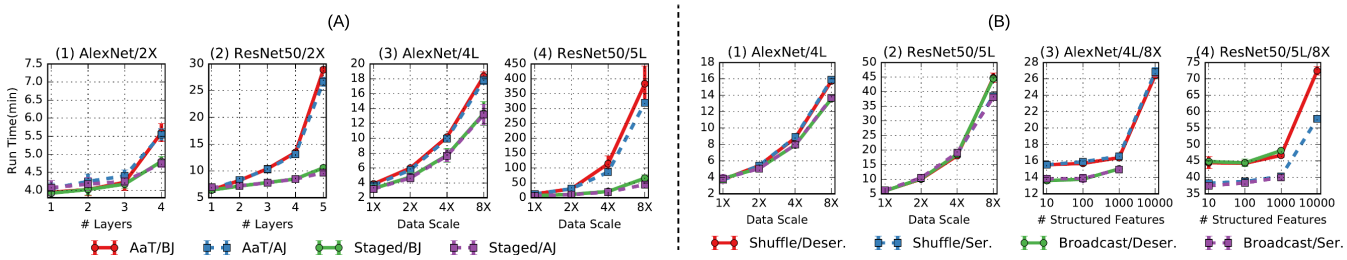

Figure 8. (A) Runtimes of logical execution plan alternatives for varying data scale and number of feature layers explored. AaT stands for All-at-a-Time. (B) Runtimes of physical plan choices for varying data scale and number of structured features.

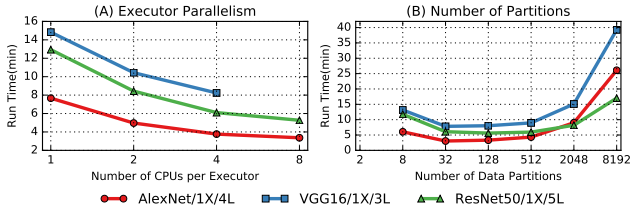

Figure 9. Varying system configuration parameters. Logical and physical plan choices are fixed to *Staged*, *After Join*, *Shuffle*, and *Deserialized*.

of JVM heap. Other parameters are set by Algorithm 1. The layers explored for each CNN are the same as before.

Logical Execution Plan Decisions. We compare four combinations: *AaT* or *Staged* combined with inference *After Join (AJ)* or *Before Join (BJ)*. We vary both $|L|$ (dropping lower layers) and data scale for AlexNet and ResNet. Figure 8(A) shows the results. 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 *AaT* plans get much slower, especially for ResNet (Figure 8(A.2,A.4)); this is due to disk spills of large intermediate data. Across the board, *AJ* plans are mostly comparable to their *BJ* counterparts but marginally faster at larger scales. The takeaway is that these results validate our choice of using only *Staged*/*AJ* in Vista, viz., Plan (E) in Figure 6 in Section 4.2.1.

Physical Plan Decisions. We compare four combinations: *Shuffle* or *Broadcast* join and *Serialized (Ser.)* or *Deserialized (Deser.)* persistence format. We vary both data scale and number of structured features ($|X_{str}|$) for both AlexNet and ResNet. The logical plan used is *Staged*/*AJ*. Figure 8(B) shows the results. On ResNet, all four plans are almost indistinguishable regardless of the data scale (Figure 8(B.2)), except at the 8X scale, when the *Ser.* plans slightly outperform the *Deser.* plans. On AlexNet, the *Broadcast* plans slightly outperform the *Shuffle* plans (Figure 8(B.1)). Figure 8(B.3) shows that this gap remains as $|X_{str}|$ increases but the *Broadcast* plans crash eventually. On ResNet, however, Figure 8(B.4) shows that both *Ser.* plans are slightly faster than their *Deser.* counterparts but the *Broadcast* plans still crash eventually. The takeaway is that no one combination is always dominant, validating the utility of an automated optimizer like ours to make these decisions.

System Configuration Decisions. We vary *cpu* and n_p, with our optimizer configuring the memory. We pick *Staged*/*AJ*/*Shuffle*/*Deser.* as the logical-physical plan combination. Figures 9(A,B) show the results for all CNNs. As explained in Section 4.3, the runtime decreases with *cpu* for all CNNs, but VGG eventually crashes (at 8 cores) due to the blowup in CNN Inference Memory. The runtime decrease with *cpu* is sub-linear though. To drill into this issue, we plot the speedup against *cpu* on 1 node for data scale 0.25X (to avoid disk spills). Figure 10(C) shows the results: the speedups plateau at 4 cores. As mentioned in Section 4.3, this is as expected, since CNN inference dominates total runtimes and TF always uses all cores regardless of *cpu*.

Figure 9(B) shows non-monotonic behaviors with n_p. At low n_p, Spark crashes due to insufficient Core Memory for the join. As n_p goes up, runtimes go down, since Spark uses more parallelism (up to 32 cores). Eventually, runtimes rise again due to Spark overheads for running too many tasks. In fact, when $n_p > 2000$, Spark compresses task status messages, leading to high overhead. The Vista optimizer (Algorithm 1) sets n_p at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, which yield close to the fastest runtimes. The

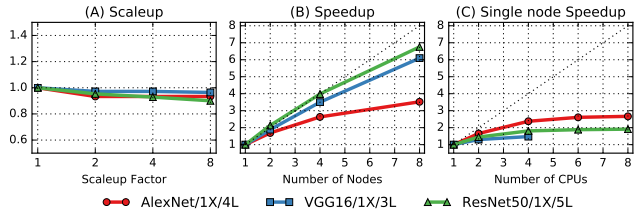

Figure 10. (A,B) Scaleup and speedup on cluster. (C) Speedup for varying *cpu* on one node with 0.25x data. Logical and physical plan choices are fixed to *Staged*, *After Join*, *Shuffle*, and *Deserialized*.

takeaway is that these settings involve non-trivial CNN-specific efficiency tradeoffs and thus, an automated optimizer like ours can free ML users from such tedious tuning.

Scalability. We evaluate the scaleup (weak scaling) and speedup (strong scaling) of the logical-physical plan combination of *Staged*/*After Join*/*Shuffle*/*Deserialized* for varying number of worker nodes (and also data scale for scaleup). While CNN inference and M are embarassingly parallel, data reads from HDFS and the join can bottleneck scalability. Figures 10 (A,B) show the results. We see near-linear scaleup for all 3 CNNs. But Figure 10 (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 [11]. 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. Thus, Spark overheads become non-trivial in AlexNet's case (more details in Appendix C of our addendum [17]).

Summary of Results. Vista reduces runtimes (even up to 10x) and avoids memory-related crashes by automatically handling the tradeoffs of logical execution plan, system configuration, and physical plan. Our new Staged execution plan offers both high efficiency and reliability. CNN-aware system configuration for memory apportioning, data partitioning, and parallelism is critical. Broadcast join marginally outperforms shuffle join but crashes at larger scales. Serialized disk spills are marginally faster than deserialized. Overall, Vista automatically optimizes such complex tradeoffs, freeing ML users to focus on their ML exploration.

5.3 Discussion of Limitations

A marriage between deep learning systems and parallel dataflow systems will be beneficial for unified analytics over structured and unstructured data. But as this paper shows, 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 currently supports one image per data example, a roster of popular CNNs, and linear models for downstream ML. Nothing in Vista makes it difficult to relax these assumptions. For instance, supporting more downstream ML models only 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 kinds of multimodal analytics too, e.g., image captioning [38]. Our focus is on integrating images with structured data. A related but orthogonal line of work is "multimodal learning" in which deep neural networks are trained from scratch on images [46, 51]; this incurs high costs for resources and labeled data, which feature transfer mitigates.

Multimedia Systems. The multimedia and database systems communities have studied "content-based" image retrieval, video retrieval, and similar queries over multimedia data [20, 37]. But they typically used non-CNN features such as SIFT and HOG [25, 43]. Such systems are orthogonal to our work, since we focus on CNN feature transfer, not retrieval queries on multimedia data. One could integrate Vista with multimedia databases.

Query Optimization. Our work is inspired by a long line of work on optimizing queries with UDFs, multi-query optimization (MQO), and self-tuning DBMSs. For instance, [24, 33] studied the problem of predicate migration for optimizing complex relational queries with joins and UDF-based predicates. Unlike such works on queries with UDFs in the `WHERE` clause, our work can be viewed as optimizing UDFs expressed in the `SELECT` clause for materializing CNN feature layers. New plans of Vista can be viewed as a form of MQO, which has been studied extensively for SQL queries [49]. Vista is the first system to bring the general idea of MQO to complex CNN feature transfer workloads. We do so by formalizing partial CNN inference operations as first-class citizens for query processing and optimization.

System Auto-tuning. There is much prior work on auto-tuning the configuration of RDBMSs, Hadoop/MapReduce, and Spark for relational workloads (e.g., [34, 53]). Our work is inspired by these works but ours is the first to focus on the CNN feature transfer workload. We explain the new efficiency and reliability issues caused by CNNs and feature layers and apply our insights for CNN-aware auto-tuning in our setting that straddles PD and DL systems.

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 integrating parallel dataflow and deep learning systems to support and optimize a key emerging workload in this context: feature transfer from deep CNNs. By enabling more declarative specification and by formalizing partial CNN inference, Vista automates much of the data management and systems-oriented complexity of this workload, thus improving system reliability and efficiency. For future work, we plan to support more general forms of CNNs and downstream ML tasks.

References

[1] Adaptive execution in spark. https://issues.apache.org/jira/browse/SPARK-9850. Accessed March 31, 2019.

[2] Apache spark: Lightning-fast cluster computing. http://spark.apache.org. Accessed March 31, 2019.

[3] Benchmarks for popular cnn models. https://github.com/jcjohnson/cnn-benchmarks. Accessed March 31, 2019.

[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 March 31, 2019.

[5] 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 March 31, 2019.

[6] Kaggle survey: The state of data science and ml. https://www.kaggle.com/surveys/2017. Accessed March 31, 2019.

[7] Models and examples built with tensorflow. https://github.com/tensorflow/models. Accessed March 31, 2019.

[8] Open food facts dataset. https://world.openfoodfacts.org/. Accessed March 31, 2019.

[9] Parallel postgres for enterprise analytics at scale. https://pivotal.io/pivotal-greenplum. Accesses January 31, 2018.

[10] Pytorch. https://pytorch.org. Accesses January 31, 2018.

[11] The small files problem of hdfs. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed March 31, 2019.

[12] Spark best practices. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Accessed March 31, 2019.

[13] Spark memory management. https://0x0fff.com/spark-memory-management/. Accessed March 31, 2019.

[14] Sparkdl: Deep learning pipelines for apache spark. https://github.com/databricks/spark-deep-learning. Accessed March 31, 2019.

[15] Tensorframes: Tensorflow wrapper for dataframes on apache spark. https://github.com/databricks/tensorframes. Accessed March 31, 2019.

[16] Transfer learning with cnns for visual recognition. http://cs231n.github.io/transfer-learning/. Accessed March 31, 2019.

[17] Vista addendum. https://www.dropbox.com/s/3i34mk6mwzsx3ks/vista-appendix.pdf?dl=0. Accessed March 31, 2019.

[18] Abadi, M., et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org; accessed December 31, 2017.

[19] Abadi, M., et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association, pp. 265–283.

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

[21] Azizpour, H., et al. Factors of transferability for a generic convnet representation. *IEEE transactions on pattern analysis and machine intelligence 38*, 9 (2016), 1790–1802.

[22] Bhuiyan, S., et al. High performance in-memory computing with apache ignite.

[23] Canziani, A., et al. An analysis of deep neural network models for practical applications. *CoRR abs/1605.07678* (2016).

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

[25] Dalal, N., and Triggs, B. 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* (2005), CVPR '05, IEEE Computer Society, pp. 886–893.

[26] Dalal, N., and Triggs, B. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005.*

IEEE Computer Society Conference on (2005), vol. 1, IEEE, pp. 886–893.

[27] Donahue, J., et al. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proceedings of the 31st International Conference on Machine Learning* (Bejing, China, 22–24 Jun 2014), E. P. Xing and T. Jebara, Eds., vol. 32 of *Proceedings of Machine Learning Research*, PMLR, pp. 647–655.

[28] Esteva, A., et al. Dermatologist-level classification of skin cancer with deep neural networks. *Nature 542*, 7639 (Jan. 2017), 115–118.

[29] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[30] Goodfellow, I., et al. *Deep Learning*. The MIT Press, 2016.

[31] He, K., et al. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016).

[32] He, R., and McAuley, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web* (2016), International World Wide Web Conferences Steering Committee, pp. 507–517.

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

[34] Herodotou, H., et al. Starfish: A self-tuning system for big data analytics. In *In CIDR* (2011), pp. 261–272.

[35] Huang, G., et al. Densely connected convolutional networks. *CoRR abs/1608.06993* (2016).

[36] Jing, Y., et al. Visual search at pinterest. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), KDD '15, ACM, pp. 1889–1898.

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

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

[39] Krizhevsky, A., et al. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

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

[41] LeCun, Y., et al. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems* (1990), pp. 396–404.

[42] Lecun, Y., et al. Deep learning. *Nature 521*, 7553 (5 2015), 436–444.

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

[44] McAuley, J., et al. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2015), ACM, pp. 43–52.

[45] Meng, X., et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research 17*, 1 (2016), 1235–1241.

[46] Ngiam, J., et al. Multimodal deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (USA, 2011), ICML'11, Omnipress, pp. 689–696.

[47] Pan, S. J., and Yang, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering 22*, 10 (2010), 1345–1359.

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

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

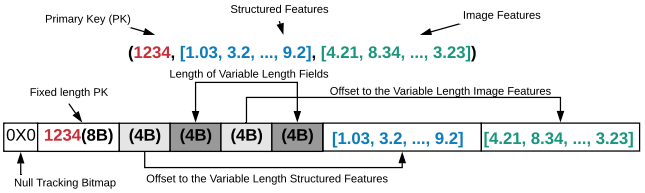

Figure 11. Spark's internal record storage format

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

[51] Srivastava, N., and Salakhutdinov, R. Multimodal learning with deep boltzmann machines. vol. 15, JMLR.org, pp. 2949–2980.

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

[53] Van Aken, D., et al. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 1009–1024.

[54] Wang, Y., and Kosinski, M. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. *Journal of Personality and Social Psychology 114* (02 2018), 246–257.

[55] Yosinski, J., et al. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (2014), NIPS'14, MIT Press, pp. 3320–3328.

[56] Zaharia, M., et al. 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* (2012), USENIX Association, pp. 2–2.

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

A Estimating Intermediate Data Sizes

We explain the size estimations in the context of Spark. Ignite also uses an internal format similar to the Spark. Spark's internal binary record format is called "Tungsten record format," shown in Figure 11. Fixed size fields (e.g., float) use 8 B. Variable size fields (e.g., arrays) have an 8 B header with 4 B for the offset and 4 B for the length of the data payload. The data payload is stored at the end of the record. An extra bit tracks null values.

Vista estimates the size of intermediate tables $T_l \; \forall l \in L$ in Figure 6(E) based on its knowledge of the CNN. 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:

$$|T_i| = \alpha_1 \times (8 + 8 + 4 \times |g_l(\hat{f}_l(I))|) + |T_{str}| \qquad (15)$$

Equation 15 assumes deserialized format; serialized (and compressed) data will be smaller. But these estimates suffice as safe upper bounds.

Figure 12 shows the estimated and actual sizes. We see that the estimates are accurate for the deserialized in-memory

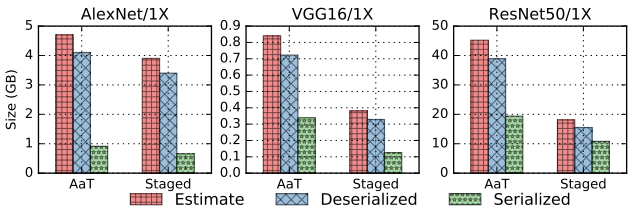

Figure 12. Size of largest intermediate table.

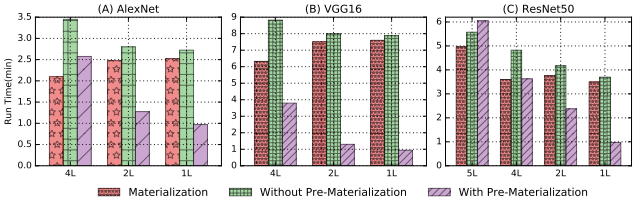

Figure 13. Runtimes comparison for using pre-materialized features from a base layer

data with a reasonable safety margin. Interestingly, *Eager* is not that much larger than *Staged* for AlexNet. This is because among its four layers explored the 4^{th} layer from the top is disproportionately large while for the other two layer sizes are more comparable. Serialized is smaller than deserialized as Spark compresses the data. Interestingly, AlexNet feature layers seem more compressible; we verified that its features had many zero values. On average, AlexNet features had only 13.0% non-zero values while VGG16's and ResNet50's had 36.1% and 35.7%, respectively.

B Pre Materializing a Base Layer

Often data scientists are interested in exploring few of the top most layers. Hence a base layer can pre-materialized before hand for later use of exploring other layers. This can save computations and thereby reduce the runtime of the CNN feature transfer workload.

However, the CNN feature layer sizes (especially for conv layers) are generally larger than the compressed image formats such as JPEG (see Table 2). This not only increases the secondary storage requirements but also increases the IO cost of the CNN feature transfer workload both when initially reading data from the disk and during join time when shuffling data over the network.

Table 2. Sizes of pre-materialized feature layers for the **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 set of experiments using the Spark-TF system to explore the effect of pre-materializing a base layer (1, 2, 4, and 5^{th} layers from top). For evaluating the ML model for the base layer no CNN inference is required. But for the other layers partial CNN inference is performed starting from the base layer using the *Staged/After Join/Deserialized/Shuffle* logical-physical plan combination. Experimental set up is same as in Section 5.2.

For AlexNet and VGG16 when materializing 4^{th}, 2^{nd}, and 1^{st} layers from the top, the materialization time increases as evaluating higher layer requires more computations (see Figure. 13 (A) and (B)). However, for ResNet50 there is a sudden drop from the materialization time of 5^{th} layer features to the materialization time of 4^{th} layer features. This can be attributed to the high disk IO overhead of writing out 5^{th} layer image features which are ~3 times larger than that of 4^{th} layer (see Figure. 13 (C)). Therefore, for ResNet50 starting from a pre-materialized feature layer, instead of raw images, may or may not decrease the overall CNN feature transfer workload runtime.

C Runtime Breakdown

We drill-down into the time breakdowns of the workloads on Spark-TF environment and explore where the bottlenecks occur. In the downstream logistic regression (LR) 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 performed 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. Input read time is dominated by reading images as there are lot of small files compared to the one big structured data file [11]. Table 3 summarizes the time breakdown for the CNN feature transfer workload. 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.

We also separately analyze the speedup behavior for the input image reading and the sum of CNN inference and LR 1^{st} iteration times (see Figure 14). When we separate out the sum of CNN inference and LR 1^{st} iteration times, we see slightly super linear speedups for ResNet50, near linear speedups for VGG16, and slightly better sub-linear speedups for AlexNet.

D Accuracy

For both Foods and a sample of Amazon (20,000 records) datasets we evaluate the downstream logistic regression model F1 score with (1) only using structured features, (2)

Table 3. Runtime breakdown for the image data read time and 1^{st} iteration of the logistic regression model (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

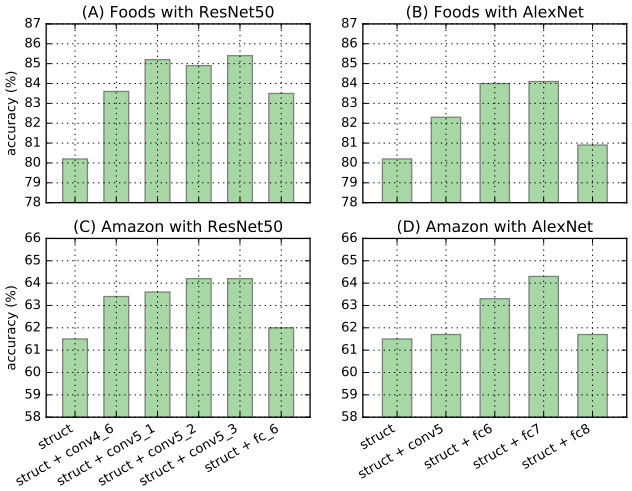

Figure 15. F1 score lifts obtained by incorporating HOG descriptors and CNN features for logistic regression model with elastic net regularization with $\alpha = 0.5$ and a regularization value of 0.01.

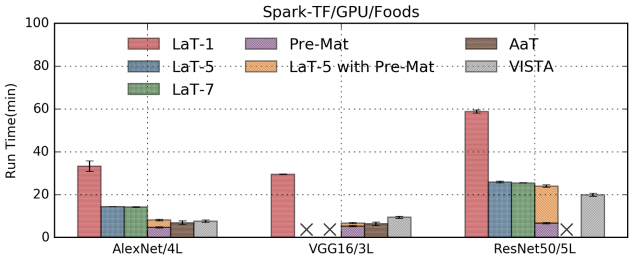

Figure 16. End-to-end reliability and efficiency on GPU. "×" indicates a system crash.

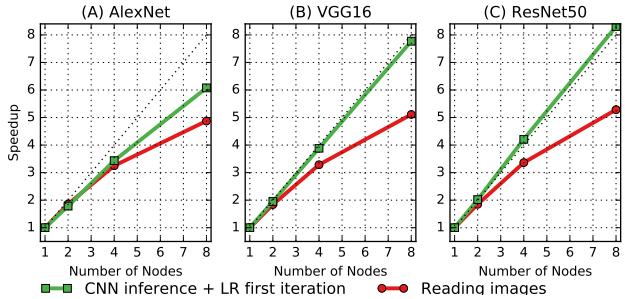

Figure 14. Drill-down analysis of Speedup Curves

Table 4. F1 scores of test datasets obtained by incorporating HOG descriptors and CNN features for logistic regression model with elastic net regularization with $\alpha = 0.5$ and a regularization value of 0.01.

	Structured Only	Structured + HOG	Structured + CNN
Foods	80.2	81.1	85.4
Amazon	61.5	62.2	64.3

structured features combined with "Histogram of Oriented Gradients (HOG)" [26] based image features, and (3) structured features combined with CNN based image features from different layers of AlexNet and ResNet models.

In all cases incorporating image features improves the classification accuracy and the improvement achieved by incorporating CNN features is higher than the improvement achieved by incorporating traditional HOG features (see Table 4 and Figure 15).

E End-to-End Reliability and Efficiency on GPUs

GPU experiments are run on Spark-TensorFlow environment using the *Foods* dataset. The experimental setup is a single node machine which has 32 GB RAM, Intel i7-6700 @ 3.40GHz CPU which has 8 cores, 1 TB Seagate ST1000DM010-2EP1 SSD, and Nvidia Titan X (Pascal) 12GB GPU. The results are shown in Figure 16. In this setup *Lazy-5* and *Lazy-7* crashes with VGG16, and *Eager* crashes with ResNet50.