VISTA: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale

Supun Nakandala
University of California, San Diego
snakanda@eng.ucsd.edu

Arun Kumar
University of California, San Diego
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 workload crashes. Experiments on real datasets show that apart from making feature transfer easier, VISTA avoids workload crashes and reduces runtimes by 58% to 92% compared to baselines.

ACM Reference Format:

1 INTRODUCTION AND MOTIVATION
Deep CNNs achieve near-human accuracy for many image analysis tasks [35, 51]. 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 [69]) are popular for structured data [4, 54], while deep learning systems (e.g., TensorFlow [22]) 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 [53]). 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 ML model (e.g., logistic regression, multi-layer perceptron, 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 [37]) 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 [36, 51], 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 increases.
Features transfer is a form of "transfer learning" that mitigates two key pains of training deep CNNs from scratch [3, 18, 56]: the number of labeled images needed is lower, often by an order of magnitude [18, 67], and the time and resource costs of training are lower, even by two orders of magnitude [3, 18]. Overall, feature transfer is now popular in many domains, including recommender systems [53], visual search [42] (product images), healthcare (tissue images) [34], nutrition science (food images) [10], 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 [18, 24, 33, 67]. But trying multiple layers becomes a bottleneck 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 decision tree-based ML models on structured data [8]; thus, a DL system alone is too limiting.

Figure 2(B) illustrates the manual process. Suppose Alice tries layer 5 (L5) to layer 7 (L7) (say) from a given CNN. She first runs CNN inference in DL system (e.g., TensorFlow) to materialize L5 for all images in her dataset. She loads this large data file with image features into PD system (e.g., Spark), joins it with the structured data, and trains a downstream multimodal ML model (e.g., using MLlib [54] or TensorFlow). She repeats this for L6 and then for L7. Apart from being tedious, 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 [37]. So, 10GB of data blows up to 560GB for just one layer! Forcing ML users to handle such large intermediate data files on in-memory PD systems can easily cause workload crashes due to exhausting available system memory. Alternatively, writing these feature files to disk and reading iteratively will incur significant overheads due to costly disk reads/writes, thus reducing efficiency further.

### 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 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 workload crashes.

---

**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).
(1) **Declarativity.** **Vista** lets users specify what CNNs and layers to try, but 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 workload 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 "Lazy" 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, "Eager" materializes all layers of interest in one go (Section 4). It avoids redundancy but is prone to memory-related crashes if the intermediate data does not fit in memory. Alternatively, enabling disk spills for the Eager plan will avoid crashes but will be inefficient due to costly disk reads/writes. 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 [25], with TensorFlow as the DL system. 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.

Our approach is inspired by the long line of work on multi-query optimization in RDBMSs [60]. But our execution plans and optimizer have no counterparts in prior work 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:

- 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%.

**Outline.** Section 2 presents some technical background. Section 3 formalizes the dataflow of the feature transfer workload, explains our assumptions, and provides an overview of **Vista**. Section 4 dives into the systems tradeoffs and presents our optimizer. Section 5 presents the experiments. We discuss related work in Section 6 and conclude in Section 7.

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 [36, 51]. 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 [50]. CNNs typically surpass older hand-crafted image features such as SIFT and HOG in accuracy [30, 52]. 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 [36].

**Transfer Learning with CNNs.** Transfer learning mitigates the cost and labeled data requirements of training deep CNNs from scratch [56]. 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 [18, 24, 33, 67]. Also, lower layer features are often much larger; so, simple feature selection such as extra pooling is typically used [24]. Such feature transfer underpins recent breakthrough applications of CNNs in detecting cancer [34], diabetic retinopathy [63], facial analysis [66], and multimodal recommendation systems [53].
Spark, Ignite, and TensorFlow. Spark and Ignite are popular distributed memory-oriented data systems [2, 25, 69]. 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 can be configured as a purely in-memory system or as an in-memory cache for disk resident data. Both systems support user-defined functions (UDFs) to let users run ML algorithms directly on large datasets, e.g., with Spark MLlib [54].

TensorFlow (TF) is a system for training and running neural networks [21]. Models in TF are specified as a “computation graph,” with nodes representing operations over “tensors” (multi-dimensional arrays) and edges representing dataflow. TensorFlow and Spark are libraries that integrate Spark and TF [15, 16]. TensorFlow 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. Furthermore, due to manual memory and system configuration tuning, it is also prone to memory-related workload crashes. Thus, both the functionality and techniques of Vista are complementary to SparkDL. Horovod [7] is a distributed SGD system for training TF computational graphs that is now integrated with Spark to enable training directly over Spark-resident data.

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 \( t \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d} \) 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. Gray-scale images have 2-dimensional tensors; colored ones, 3-dimensional (with RGB pixel values). We now define some abstract data types 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 \( t' \) if 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 \( n_i \) indexed TensorOps, denoted \( f() \equiv f_{n_i}(\ldots f_2(f_1())\ldots) \), wherein each TensorOp \( f_i \) is called a layer and \( n_i \) is the number of layers.\(^1\)

We use \( f_i \) to denote \( f_i(\ldots f_2(f_1())\ldots) \).

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

**Definition 3.6.** CNN inference. Given a CNN \( f \) and a shape-compatible image tensor \( t \), CNN inference is the process of computing \( f(t) \).

**Definition 3.7.** 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 \( f_{i \rightarrow j} \).

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 \( T_{str}(ID, X) \) and \( T_{img}(ID, I) \), where \( ID \) is the primary key (identifier), \( X \in \mathbb{R}^{d_x} \) is the structured feature vector (with \( d_x \) features, including label), and \( I \) are raw images (say, as files on HDFS). We are also given a CNN \( f \) with \( n_i \) layers, a set of layer indices \( L \subseteq [n_i] \) 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:

\[
\forall l \in L: \quad (1) \quad T'_{img, l}(ID, g_l(\hat{f}_l(I))) \leftarrow \text{Apply } (g_l \circ \hat{f}_l) \text{ to } T_{img} \tag{2} \\
T'_l(ID, X'_l) \leftarrow T_{str} \Rightarrow T'_{img, l} \tag{3} \\
\text{Train } M \text{ on } T'_l \text{ with } X'_l \equiv [X, g_l(\hat{f}_l(I))] \tag{4}
\]

\(^1\)We use sequential (chain) CNNs for simplicity of exposition; it is easy to extend our definitions to DAG-structured CNNs such as DenseNet [41].
Step (2) runs partial CNN inference to materialize layer \( l \) and flattens it with \( g_i \), a shape-compatible FlattenOp. 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 \( g \) to reduce dimensionality for \( M \) [24]. 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 Lazy. This plan is cumbersome, inefficient due to redundant CNN inference, and/or prone to workload crashes due to inadvertently mismanaged memory. Our goal is to resolve these issues. Our approach is to elevate this workload to a declarative level, obviate manual feature transfer, automatically reuse intermediate results, and optimize the system configuration and execution for better reliability and efficiency.

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 [46], VGG16 [61], and ResNet50 [37] due to their popularity in real feature transfer applications [53, 66]. Second, we support only one image per data record. We leave support for arbitrary CNNs and multiple images per example to future work. 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, and (3) the Vista optimizer. Our Python API 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, hyperparameter tuning, and model artifacts. Fourth, data tables \( T_{str} \) and \( T_{img} \) and statistics about the data. We provide detailed API information in our technical report [19].

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., \( f, f_l, g_l \), and \( f_{l-1} \) for the CNNs in its roster (currently, AlexNet, VGG16, and ResNet50). These UDFs specify the computational graphs for TF and invoke Spark’s DataFrame and TensorFlow APIs with appropriate inputs based on our optimizer’s decisions. Image and feature tensors are stored with our custom TensorList datatype. Finally, Vista invokes downstream ML model training on the concatenated feature vector and obtains \( |L| \) trained downstream models. Overall, Vista frees ML users from manually writing TF code for CNN feature transfer, saving features as files, performing joins, or tuning Spark for running this workload at scale.

4 TRAFOFFS 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 the three dimensions of efficiency-reliability tradeoffs and apply our analyses to design the Vista optimizer.

4.1 Memory Use Characterization

It is important to understand and optimize the memory use behavior of the feature transfer workload, since mismanaged memory can cause frustrating workload 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/DL 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 4(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, 13, 14]. OS Reserved Memory is typically a few GBs. Our workload requires rethinking memory apportioning due to interesting new issues caused by deep CNN models, (partial) CNN inference, feature layers, and the downstream ML task.

(1) The guideline of using most of System Memory for Storage and Execution no longer holds. In both Spark and Ignite, CNN inference in DL system (e.g., TF) uses System Memory outside Storage and Execution regions. If a DL model is used as the downstream ML model, it will also use memory outside of Storage and Execution regions. The memory footprint of DL models is non-trivial. For parallel query execution in PD systems, each execution thread will spawn its own DL model replica, multiplying the footprint.

(2) Many temporary objects are created when reading serialized DL models to initialize the DL system, for buffers to read inputs, and to hold features created by CNN inference. All 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 tedious for ML users.

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

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.

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.

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

(1) DL Execution Memory blowups. Serialized file formats of CNNs and downstream ML models often underestimate their in-memory footprints. Along with the replication by multiple threads, DL Execution 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, downstream ML models, 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 PD 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. For the downstream ML model, the Driver may also have 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 5(A) illustrates the Lazy 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 5(B), the shuffle costs of the join will go down. We call this slightly modified plan **Lazy-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 Eager plan shown in Figure 5(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 (concatenated with $X$) projected from the `TensorList`. Eager-Reordered, shown in Figure 5(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 workload 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 5(E). It splits partial CNN inference across the layers in $L$ and invokes $M$ on branches 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 Eager and Eager-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 an architecture is unlikely to yield high accuracy, since it loses too much information too soon [36]. Indeed, no popular deep CNN has such an architecture. Based on our above analysis, we find that it suffices for Vista to only use our new Staged plan (validated in Section 5).

#### 4.2.2 System Configuration Tradeoffs

Logical plans are generic and independent of the PD system. 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. While the tuning of such parameters is well understood for SQL and MapReduce [40, 64], 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

---

2In our current prototypes, every TF invocation by a worker uses all cores regardless of how many cores are assigned to that worker. But, one TF invocation per used core increases overall throughput and reduces runtimes.
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 (|Tstr|), the images table (|Timg|), and all intermediate data tables (|Ti| for i ∈ L) shown in Figure 5(E). Vista also looks up the CNN’s serialized size |fser|, runtime memory footprint |fmem|, and runtime GPU memory footprint |f_cpu_gpus| from its roster, in which we store these statistics. Then, Vista calculates the runtime memory footprint of the downstream model |Mimg| and the largest total number of features (based on L). For instance, for logistic regression, |M| is proportional to the sum of structured features and the maximum number of CNN features for any layer (max g(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:

\[
s_{\text{single}} = \max_{1 \leq i \leq L} |T_i| \quad (5)
\]

\[
s_{\text{double}} = \max_{1 \leq i \leq L} (|T_i| + |T_{i+1}|) - |T_{str}| \quad (6)
\]

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 memstorage. Raising cpu increases parallelism, which could reduce runtimes. But it also raises the DL Execution Memory needed, which forces memstorage to be reduced, thus increasing potential disk spills/cache misses for Ti’s and raising runtimes. This tension is captured by the following objective function:

\[
\min_{\text{cpu}, \text{memstorage}} \frac{r + \max(0, s_{\text{double}} - s_{\text{single}})}{\text{cpu}} \quad (7)
\]

In the numerator, r captures the total compute and communication costs, which are effectively “constant” for this optimization. The second term captures disk spill costs for Ti’s. The denominator captures the degree of parallelism. While this objective is ideal, it is impractical and needlessly complicated for our purposes due to three reasons. (1) Estimating r is tedious, since it involves join costs, data loading costs, etc. (2) More importantly, we hit a point of diminishing returns with cpu quickly, since CNN inference typically dominates

![Diagram](image-url)
Overall, our insights above yield a simpler objective that is still a reasonable surrogate for minimizing runtimes:

\[
\max_{\text{cpu}, n_p, \text{mem}_{\text{storage}}} \text{cpu} \tag{8}
\]

The constraints for the optimization are as follows:

\[
1 \leq \text{cpu} \leq \min\{\text{cpu}_{\text{sys}}, \text{cpu}_{\text{max}}\} - 1 \tag{9}
\]

total runtime and DL systems like TF, anyway uses all cores regardless of cpu. That is, this workload’s speedup against cpu will be quite sub-linear (confirmed by Figure 12(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 [13, 15]. Thus, we suffice; interestingly, a similar observation is made in Spark.

Section 5). Empirically, we find that about 7 cores typically suffice; interestingly, a similar observation is made in Spark guidelines for purely relational workloads [13, 15]. Thus, we suffice; interestingly, a similar observation is made in Spark.

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

\[
\max_{\text{cpu}, n_p, \text{mem}_{\text{storage}}} \text{cpu} \tag{8}
\]

The constraints for the optimization are as follows:

\[
1 \leq \text{cpu} \leq \min\{\text{cpu}_{\text{sys}}, \text{cpu}_{\text{max}}\} - 1 \tag{9}
\]
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 \textit{join} to broadcast if the predefined maximum broadcast data size constraint is satisfied; otherwise, we set it to \textit{shuffle}. Finally, as per Section 4.2.3, \textit{pers} is set to \textit{serialized}, if disk spills/cache misses are likely (based on the newly set \textit{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.

5 EXPERIMENTAL EVALUATION

We empirically validate if \textsc{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: \textit{Foods} [10] and \textit{Amazon} [38]. \textit{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. \textit{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 [49]. We convert the indicator vector of categories to 100 numeric features using PCA. All images are resized to $227 \times 227$ resolution, as needed by popular CNNs. Overall, \textit{Foods} is about 300 MB in size; \textit{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 web page.

Workloads. We use three ImageNet-trained deep CNNs: AlexNet [46], VGG16 [61], and ResNet50 [37], obtained from TF model zoo [9]. They complement each other in terms of model size [26]. 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 [37]). Following standard practices [18, 67], we apply max pooling on convolutional feature layers to reduce their dimensionality before using them for $M^4$.

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 [57]. 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 TensorFlow v1.2.0, 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 memory-only mode; each node runs one worker. All runtimes reported are the average of three runs with 90% confidence intervals. We chose these systems due to their popularity but the takeaways from our experiments are applicable to other DL systems (e.g., PyTorch [12]) and PD systems (e.g., Greenplum [11]) as well.

5.1 End-to-End Reliability and Efficiency

We compare \textsc{Vista} with five baselines: three naive and two strong. Lazy-1 (1 CPU per Executor), Lazy-5 (5 CPUs), and Lazy-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, 13] (29 GB JVM heap, shuffle join, deserialized, and defaults for all other parameters, including \textit{np} and memory apportioning). Ignite is configured with a 4 GB JVM heap, 25 GB off-heap Storage Memory, and \textit{np} set to the default 1024. Lazy-5 with Pre-mat and Eager are strong baselines based on our tradeoff analyses in Section 4.2.1. In Lazy-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

\footnote{Filter width and stride for max pooling are set to reduce the feature tensor to a $2 \times 2$ grid of the same depth.}

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{OptimizeFeatureTransfer}{ }
\State \textbf{inputs}: see Table 1(A)
\State \textbf{outputs}: see Table 1(B)
\For{$x = \min\{\text{cpu\_sys}, \text{cpu\_max}\} - 1$ to 1 $\times$ Linear search}
\State $n_p \leftarrow \text{NumPartitions}(|\text{single}\_x, n_{\text{nodes}}|)$
\State $\text{mem\_worker} \leftarrow \text{mem\_sys} - \text{mem\_os\_rstv} - x \times |f|$ mem\_user
\State $\text{mem\_user} \leftarrow \max(|f|_{\text{ser}} + x \times \alpha \times |\text{single}/n_p|, x \times |M|_{\text{mem}})$
\If{$\text{mem\_worker} > \text{mem\_user}$}
\State $\text{cpu} \leftarrow x$
\State $\text{mem\_storage} \leftarrow \text{mem\_worker} - \text{mem\_user} - \text{mem\_core}$
\State $\text{join} \leftarrow \text{shuffle}$
\If{$|T|_{\text{ser}} < b_{\text{max}}$}
\State $\text{join} \leftarrow \text{broadcast}$
\EndIf
\If{$|\text{pers}\_s\_ser| < s_{\text{double}}$}
\State $\text{pers} \leftarrow \text{serialized}$
\EndIf
\State $\text{return}\ (\text{mem\_storage}, \text{mem\_user}, \text{cpu}, n_p, \text{join}, \text{pers})$
\EndIf
\EndFor
\EndProcedure
\Procedure{NumPartitions}{$|\text{single}\_x, n_{\text{nodes}}|$}
\State $\text{totalcores} \leftarrow x \times n_{\text{nodes}}$
\State $\text{singleton} \leftarrow |\text{max}\times \text{totalcores}| \times |\text{totalcores}|
\EndProcedure
\end{algorithmic}
\end{algorithm}
we explicitly apportion CNN Inference memory, Storage Vista workload crash. (B) Comparing TFT+Beam vs. 72% faster; compared to 4.2.1. However, when the size of the intermediate data does Lazy-7 on further ML analysis. Compared to level systems issues (Section 4)–time they can now spend but that forces ML users to waste time wrestling with low- because the layers of AlexNet and ResNet are much larger than the images, which raises data I/O and join costs.

![Figure 6](image6.png)

Figure 6: End-to-end reliability and efficiency. "×" means the workload crashed. Overall, Vista offers the best or near-best runtimes and never crashes, while the alternatives are much slower or crash in some cases.

time spent on pre-materializing the lowest layer specified. Eager is an alternative plan explained in Section 4.2.1; we use 5 CPUs per Executor. For Lazy-5 with Pre-mat and Eager, we explicitly apportion CNN Inference memory, Storage Memory, User Memory, and Core Memory to avoid workload crashes. Note that Lazy-5 with Pre-mat and Eager actually need parts of our code from Vista. As for M, we run logistic regression for 10 iterations. Figure 6 presents the results.

Overall, Vista improves reliability and/or efficiency across the board. On Spark-TF, Lazy-5 and Lazy-7 crash on both datasets for VGG16. On Ignite-TF, Lazy-7 crashes for all CNNs on Amazon, while for ResNet50, Lazy-7 on Foods also crashes. These are due to memory related crash scenarios explained in Section 4.1. On Ignite-TF, Eager on Amazon also crashes for ResNet50 due to intermediate data exhausting the total available system memory. When Eager does not crash and the intermediate data fits in memory, its efficiency is comparable to Vista, which validates our analysis in Section 4.2.1. However, when the size of the intermediate data does not fit in memory, as with Amazon on Spark for ResNet50, Eager incurs significant overheads due to costly disk spills. Lazy-5 with Pre-mat does not crash, but its runtimes are comparable to Lazy-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.

More careful tuning could avoid the crashes with Lazy. But that forces ML users to waste time wrestling with low-level systems issues (Section 4)–time they can now spend on further ML analysis. Compared to Lazy-7, Vista is 62%–72% faster; compared to Lazy-1, 58%–92%. 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 tried, the more redundancy there is and the faster Vista will be.

Experiments on a GPU. We ran GPU experiments on Spark-TensorFlow environment using the Foods dataset. Experimental setup is a single node machine with 32 GB RAM, Intel i7-6700 @ 3.40GHz CPU with 8 cores, 1 TB Seagate ST1000DM010-2EP1 SSD, and Nvidia Titan X (Pascal) 12GB GPU. Figure 7 presents the results. In this setup Lazy-5 and Lazy-7 crash with VGG16. For ResNet50, Eager takes significantly more time to complete compared to Vista due to costly disk spills. Overall, the experimental results on both CPU and GPU settings confirm the benefits of an automatic optimizer such as ours for improving reliability and efficiency, which could reduce both user frustration and costs.

Comparing against TF Transform+Beam. TensorFlow Transform (TFT) [6] is a library for pre-processing input data for TF. It can wrap CNN models as pre-processing functions and be run on Apache Beam at scale to generate ML-ready features in TFRecord format. So, TFT is akin to the role of TensorFrames in Vista. We compare TFT+Beam against Vista on Foods/ResNet50 with varying number of layers explored. For TFT+Beam, we first join the structured data with images and then extract and write out features for all the layers in one go (similar to our Eager plan). We then train a 3-layer MLP (each hidden layer has 1024 units) for 10 iterations using distributed TF/Horovod. We use Apache Flink (v1.9.2) as the backend runtime for Beam. For Flink, through trial and error, we chose a working configuration with a parallelism of 32, JVM heap of 25GB, and User Memory fraction of 60% (default 30%). Increasing the parallelism, reducing the heap size, and/or User Memory fraction resulted in various memory-related crashes. For Vista we use Spark backend and use TF/Horovod to train the same downstream MLP. Figure 7 (B) presents the results.

When exploring only the last layer, TFT+Beam is slightly faster than Vista. However, when exploring more layers, Vista starts to clearly outperform TFT+Beam. This is because
extraction of all the layers in one go puts significant memory pressure on Flink, causing costly disk spills. VISTA’s staged materialization plan keeps the memory pressure to a minimum. The fact that we had to manually figure out a working configuration for Flink to run this workload underscores the importance of automatically tuning these parameters without any user intervention. It also shows that the trade-offs discussed in Section 4.1 are generally applicable when integrating DL and PD systems.

5.2 Accuracy

All approaches in Figure 6 (including VISTA) yield identical downstream models (and thus, same accuracy) for a given CNN layer. For both Foods and a sample of Amazon (20,000 records) datasets we evaluate the downstream logistic regression model test F1 score with (1) only using structured features, (2) structured features combined with “Histogram of Oriented Gradients” (HOG) [31] base image features, and (3) structured features combined with CNN based image features from different layers of AlexNet and ResNet. Figure 8 presents the results.

In all cases incorporating image features improves the classification accuracy, and CNN features offer significantly higher lift in accuracy than traditional HOG features. We saw F1 score lifts of 3% to 5% for the logistic regression model with feature transfer on test datasets (20% of the data). As expected, the lift varies across CNNs and layers. For instance, on Foods, structured features alone give 80.2% F1 score. 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. These results reaffirm the need to try multiple layers and thus, the need for a system such as VISTA to simplify and speed up this process. We also tried a decision tree as the downstream ML model. For Foods it yielded a test F1 score of 88.5% and for Amazon an F1 score of 61.4%. However, in both cases incorporating CNN features didn’t improve the accuracy significantly. We believe this is because the depths of the conventional decision tree models are not large enough to reap the benefits of CNN features. We leave the analysis on the suitability of different ML models for CNN feature transfer for future work as it is orthogonal to our work.

5.3 Drill-Down Analysis of Tradeoffs

We now analyze how VISTA navigates the tradeoffs explained in Section 4 using Spark-TF prototype of VISTA. 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% 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: Eager 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 9 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 Eager plans get much slower, especially for ResNet (Figure 9(2,4)); 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 5 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 (|Xstr|) for both AlexNet and ResNet. The logical plan used is Staged/AJ. Figure 10 shows the results. On ResNet, all four plans are almost indistinguishable regardless of the data scale (Figure 10(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 10(1)). Figure 10(3) shows that this gap remains as |Xstr| increases but the Broadcast plans crash eventually. On ResNet, however, Figure 10(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.

Figure 8: Test F1 scores for various sets of features for training a logistic regression model with elastic net regularization with \( \alpha = 0.5 \) and a regularization value of 0.01.
Optimizer Correctness. We vary cpu and np while explicitly apportioning the memory regions based on the chosen cpu value. We pick Staged/AJ/Shuffle/Deser. as the logical-physical plan combination. Figures 11(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 (beyond 4 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 12(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. Overall, we can see that the Vista optimizer (Algorithm 1) picks either optimal or near-optimal cpu values; AlexNet: 7, VGG16: 4, and ResNet50: 7.

Figure 11(B) shows non-monotonic behaviors with np. At low np, Spark crashes due to insufficient Core Memory for the join. As np 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 np > 2000, Spark compresses task status messages, leading to high overhead. The Vista optimizer (Algorithm 1) sets np at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, which yield close to the fastest runtimes. The 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/AJ/Join/Shuffle/Deserialized for varying number of worker nodes (and also data scale for scaleup). While CNN inference and M are embarrassingly parallel, data reads from HDFS and the join can bottleneck scalability. Figures 12(A,B) show the results. We see near-linear scaleup for all 3 CNNs. But Figure 12 (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 [17]. 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.

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.

5.4 Discussion of Limitations

Deep integration between PD and DL systems. Vista currently supports one image per data example and a roster of popular CNNs. Nothing in Vista makes it difficult to relax these assumptions. For instance, supporting arbitrary CNNs requires static analysis of TF computational graphs. We leave such extensions to future work.

Feature transfer from other types of neural models. Apart from image analytics, natural language processing (NLP) is another domain where feature transfer from pre-trained models has proven to be useful [58]. BERT [32] is one such popular pre-trained NLP model. It has a DAG style encoder-decoder architecture where as CNNs can be simplified into a sequence of logical layers. Similar to CNNs, in order to pick the best layer one has to explore multiple decoder layer outputs. Also, aggregating features from multiple decoder layers using concatenation or element-wise addition is also common. Thus, a feature layer in BERT depends on multiple input layers and supporting it in Vista requires generalizing our staged materialization plan to support arbitrary DAG architectures. 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 [44]. 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 [55, 62]; 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 [23, 43]. But they typically used non-CNN features such as SIFT and HOG [30, 52] not learned or hierarchical CNN features, although there is a resurgence of interest in CBIR with CNN features [65, 68]. 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, [20, 27, 39] studied the problem of optimizing complex relational queries with 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. Vista is the first system to bring the general idea of MQO to complex CNN feature transfer workloads, which has been studied extensively for SQL queries [60]. We do so by formalizing partial CNN inference operations as first-class citizens for query processing. In doing so, our work expands a recent line of work on materialization optimizations for feature selection in linear models [45, 71] and integrating ML with relational joins [28, 47, 48, 59]. Finally, our work also expands the work in database systems on optimizing memory usage based on data access patterns of queries [29]. But ours is the first work to study this issue in depth for CNN feature transfer queries.

System Auto-tuning. There is much prior work on auto-tuning the configuration of RDBMSs, Hadoop/MapReduce, and Spark for relational workloads (e.g., [40, 64]). 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 new opportunities for exploiting images and other unstructured data 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. 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, downstream ML tasks, and other types of unstructured data as well.

Acknowledgments. This work was supported in part by a Hellman Fellowship. We thank the members of UC San Diego’s Database Lab and Center for Networked Systems, as well as Julian McAuley, Vijay Chidambaram, Geoff Voelker, and Matei Zaharia for their feedback on this work.
REFERENCES


[35] Konid, P., Kumar, A., Re, C., and Sashikanth, V. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data
Analytics System. PVLDB 6, 12 (2013), 1306–1309.


Figure 13: VISTA API and sample usage showing values for the input parameters and invocation.

Figure 14: Spark’s internal record storage format

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 14. 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 $T_i \forall i \in L$ in Figure 5(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 $|T_l|$ with feature layer $l = L[i]$, we have:

$$|T_l| = a_1 \times (8 + 4 \times |g_l(f_l(I))) + |T_{str}|$$  \hspace{1cm} (16)

Equation 16 assumes deserialized format; serialized (and compressed) data will be smaller. But these estimates suffice as safe upper bounds.

Figure 15 shows the estimated and actual sizes. We see that the estimates are accurate for the deserialized in-memory 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 4th 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.

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 5th 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 5.3.

**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 [17]. 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 1st iteration on the first layer (e.g 5th layer from top for ResNet50) where the CNN inference has to be performed starting from raw images.
Table 3: Runtime breakdown for the image data read time and 1st iteration of the logistic regression model (Layer indices starts from the top and runtimes are in minutes).

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

Figure 17: Drill-down analysis of Speedup Curves

We also separately analyze the speedup behavior for the input image reading and the sum of CNN inference and LR 1st iteration times (see Figure 17). When we separate out the sum of CNN inference and LR 1st iteration times, we see slightly super linear speedups for ResNet50, near linear speedups for VGG16, and slightly better sub-linear speedups for AlexNet.