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

ABSTRACT

Deep convolutional neural networks (CNNs) achieve near-human accuracy on many image understanding tasks. Thus they are now increasingly used to integrate images with structured data for *multimodal analytics* applications. Since training deep CNNs from scratch is expensive, *transfer learning* has become popular: using a pre-trained CNN, one "reads off" a certain layer of CNN features to represent images and combines them with other features for a downstream ML task. Since no single layer will always offer best accuracy in general, such *feature transfer* requires comparing many layers. The current dominant approach to this process on top of scalable analytics systems such as Spark using deep learning toolkits such as TensorFlow is fraught with inefficiency due to redundant CNN inference and the potential for system crashes due to mismanaged memory. We present Vista, the first data system to mitigate such issues by elevating the feature transfer workload to a declarative level and formalizing the data model of CNN inference. Vista enables automated optimization of *feature materialization trade-offs*, distributed memory management, and system configuration. Real-world experiments show that apart from enabling seamless feature transfer, Vista substantially improves system reliability and reduces runtimes by up to 90%.

ACM Reference Format:
. 2018. Materialization Trade-offs for Feature Transfer from Deep CNNs for Multimodal Data Analytics. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep convolutional neural networks (CNNs) have revolutionized computer vision, yielding near-human accuracy for many image understanding tasks [51]. The key technical reason for their success is how they extract a hierarchy of parametrized features from images, with the parameters learned automatically during training [34]. Each layer of features captures a different level of abstraction about the image, e.g., low-level edges and patterns in the lowest layers to abstract object shapes in the highest layers. This remarkable ability of deep CNNs is illustrated in Figure 1(A).

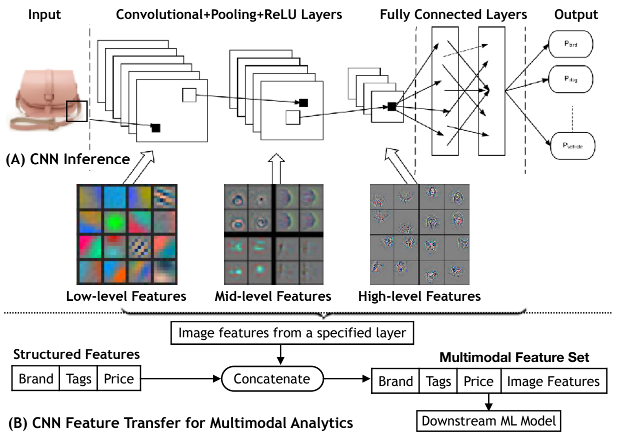

Figure 1: (A) Simplified illustration of a typical deep CNN and its hierarchy of learned features (based on [68]). (B) Illustration of CNN feature transfer for multimodal analytics.

The success of deep CNNs presents an exciting opportunity to holistically integrate image data into traditional data analytics applications in the enterprise, healthcare, Web, and other domains that have hitherto relied mainly on structured data features but had auxiliary images that were not exploited. For instance, product recommendation systems are powered by ML algorithms that relied mainly on structured data features such as price, vendor, purchase history, etc. Such applications are increasingly using CNNs to exploit product images by extracting visually-relevant features to help improve ML accuracy, especially for products such as clothing and footwear [53]. Indeed, such CNN-based feature extraction already powers visual search and analytics at some Web companies [40]. Numerous other applications could also benefit from such *multimodal analytics*, including inventory management, healthcare, and online advertising.

Since training deep CNNs from scratch is expensive in terms of resource costs (e.g., one might need many GPUs [3]) and the number of labeled examples needed, an increasingly popular paradigm for handling images is *transfer learning* [56]. Essentially, one uses a pre-trained deep CNN, e.g., ImageNet-trained AlexNet [31, 45] and "reads off" a certain layer of the features it produces on an image as the image's representation [17, 32]. Any downstream ML model can use these image features along with the structured features, say, the popular logistic regression model or even shallow neural networks. Figure 1(B) illustrates this process. Thus, such *feature transfer* helps reduce costs dramatically for using deep CNNs. Indeed, this paradigm is responsible for many high-profile successes of CNNs, including detecting cancer [33] and diabetic retinopathy [62], facial analyses [19], and product recommendations and search [40, 53].

Alas, feature transfer creates a new practical bottleneck for data scientists: it is impossible to say in general which layer of a CNN will yield the best accuracy for the downstream ML task [23]. The rule of thumb is to extract and compare multiple layers [23, 65]. This is a *model selection* process that combines CNN features with structured data [47]. It requires training the downstream ML model for each CNN layer of interest. Surprisingly, the current dominant approach to this process is to manually *materialize* each CNN layer from scratch as flat files using tools such as TensorFlow [21] and loading such data into a scalable analytics system for the downstream ML task, say, with Spark and MLlib [54], which is increasingly popular among enterprises [4, 8]. Such manual management of files of features and system memory could frustrate users and reduce productivity. Furthermore, as we will show, this approach also ignored opportunities to avoid redundant computations, which wastes runtimes and raises costs, especially in the cloud.

In this paper, we resolve the above issues for scalable feature transfer from deep CNNs for multimodal analytics. We start with a simple but crucial observation: the various layers of a typical CNN are *not* independent–*extracting a higher layer requires a superset of the computations needed for a lower layer.* This observation leads us to a classical database systems-style concern: *view materialization trade-offs.* In database parlance, the current approach of materializing all layers of interest from scratch on demand is a form of *lazy materialization*, where a "view" is a layer of CNN features. This approach wastes runtime due to redundancy in CNN inference computations across layers of interest.

One might then ask: *Why not materialize and cache all layers of interest in one go?* This is a form of *eager materialization.* While it reduces runtimes, this alternative approach increases *memory pressure*, since CNN features are often much larger than the input. For example, one of ResNet50's layers is 784kB, while the input image is 14kB [35]; so, 10GB of images will blow up to 560GB of features! Such data blowups lead to non-trivial systems trade-offs for *balancing runtimes and memory/storage space.* Performed naively, eager materialization can cause *system crashes*, which could frustrate users and raise costs again by requiring them to manually tweak the system or use needlessly more expensive machines. Caching more layers than needed can also cause *disk spills*, raising runtimes further. Thus, overall, scalable feature transfer is technically challenging due to two simultaneous systems concerns: *efficiency* (reducing runtimes) and *reliability* (avoiding system crashes).

Resolving the above dichotomy between lazy and eager materialization requires navigating complex materialization trade-offs involving feature storage, memory usage, and runtimes. Since such trade-offs are likely too low-level for most ML-oriented data scientists, we present a novel "declarative" data system to handle such trade-offs and let data scientists focus on *what* layers they want to explore rather than *how* to run this workload. We formalize the dataflow of CNN inference operations and perform a comprehensive analysis of the *abstract memory usage behavior* of this workload, inspired by work on optimizing memory usage in RDBMSs [28].

Using our analysis, we delineate *three general dimensions of systems trade-offs* for this workload. First, we compare new *logical execution plan* choices to avoid redundant CNN inference and ease memory pressure. In particular, we introduce a novel CNN-aware execution plan in between lazy and eager that is inspired by multi-query optimization from the relational query optimization literature [59]. Second, we analyze the trade-offs of key *system configuration* parameters, in particular, memory apportioning, multi-core parallelism, and data partitioning. Third, we analyze the trade-offs of two *physical execution plan* choices, viz., join operator selection and data serialization. Finally, we put together all our analyses to design an *automated optimizer* that navigates all trade-offs and picks an end-to-end system configuration and execution plan to improve both reliability and efficiency.

We prototype our ideas as a system we call Vista on top of Spark [67] and Ignite [24], two popular distributed memory-oriented data systems. This lets us piggyback on them for orthogonal benefits such as scalability and fault tolerance. We use TensorFlow for efficient CNN inference. Vista offers a high-level API for users to specify the feature transfer workload and issues queries to the underlying data system based on the input parameters and its optimizer's decisions. While we focus on Spark, Ignite, and TensorFlow due to their popularity, our ideas are generic and orthogonal to the specific systems used. One could replace Spark with Hadoop or TensorFlow with PyTorch, but still benefit from our analysis and optimization of the materialization trade-offs.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study the materialization trade-offs of scalable CNN feature transfer for multimodal analytics over image and structured data from a systems standpoint.

- We analyze the abstract memory usage behavior of this workload, delineate three dimensions of trade-offs (logical execution plan, system configuration, and physical execution plan), and present a new multi-query optimized CNN-aware execution plan.
- We devise an automated optimizer to handle all systems trade-offs and build a prototype system Vista on top of popular scalable data systems and deep learning tools to let data scientists focus on their ML exploration instead of being bogged down by systems issues.
- We present an extensive empirical evaluation of the reliability and efficiency of Vista using real-world datasets and deep CNNs and also analyze how it navigates the trade-off space. Overall, Vista detects and avoids many crash scenarios and reduces runtimes by up to 90%.

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

2 BACKGROUND

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

Deep CNNs. CNNs are a type of neural networks specialized for image data [34, 51]. They exploit spatial locality of information in image pixels to construct a hierarchy of parametric feature extractors and transformers organized as layers of various types: *convolutions*, which use image filters from graphics, except with variable filter weights, to extract features; *pooling*, which subsamples features in a spatial locality-aware way; *non-linearity* to apply a non-linear function (e.g., ReLU) to all features; and *fully connected*, which is a collection of perceptrons. A "deep" CNN just stacks such layers many times over. All parameters are trained end-to-end using backpropagation [50]. This learning-based approach to feature engineering enables CNNs to automatically construct a hierarchy of relevant image features (see Figure 1) and surpass the accuracy of prior art that relied on fixed hand-crafted features such as SIFT and HOG [29, 52]. Our work is orthogonal to how CNNs are designed, but we note that training them *from scratch* incurs massive costs: they often need many GPUs for reasonable training times [3], as well as huge labeled datasets and hyper-parameter tuning to avoid overfitting [34].

Transfer Learning with CNNs. Transfer learning is a popular paradigm to mitigate the cost and data issues with training deep CNNs from scratch [56]. One uses a pre-trained CNN, say, ImageNet-trained AlexNet obtained from a "model zoo" [5, 10], removes its last few layers, and uses it as an image feature extractor. This "transfers" knowledge learned by AlexNet to the target prediction task. If the same CNN architecture is used and the last few layers are retrained, it is called "fine tuning," but one can also use a more interpretable model such as logistic regression for the target ML task. Such transfer learning underpins recent breakthroughs in detecting cancer [33], diabetic retinopathy [62], face recognition-based analyses [19], and multimodal recommendation algorithms combining images and structured data [53]. However, no single layer is universally best for accuracy; the guideline is that the "more similar" the target task is to ImageNet, the better the higher layers will likely be [17, 23, 32, 65]. Also, lower layer features are often much larger; so, simple feature selection such as extra pooling is typically helpful [23]. Overall, data scientists have to explore at least a few layers for best results [23, 65].

Spark, Ignite, and TensorFlow. Spark and Ignite are popular distributed memory-oriented data systems [2, 24, 67]. At their core, both have a distributed collection of key-value pairs as the data abstraction. They support numerous dataflow operations, including relational operations and MapReduce. In Spark, this distributed collection (called a Resilient Distributed Dataset or RDD) is immutable, while in Ignite, it is mutable. Spark holds the data in memory and support disk spills. It uses HDFS for persistent storage. Ignite has its own native storage layer and uses the memory to operate as a cache for the data in its persistent storage. Both systems have extension capabilities in the form of user-defined functions (UDFs) that let users run ML algorithms directly on large datasets that reside in such systems. MLlib is a library of popular ML algorithms implemented over Spark; it is popular for scalable ML over structured data [4, 9].

TensorFlow (TF) is a tool for expressing ML algorithms, especially complex neural network architectures (including deep CNNs) [20, 21]. Models in TF are specified as a "computational graph," with nodes representing operations over "tensors" (multi-dimensional arrays) and edges representing dataflow. To execute a graph, one selects a node to "run" after specifying all its input data. *TensorFrames* and *SparkDL* are APIs that integrate Spark and TF [15, 16]. They enable the use of TF within Spark by invoking TF sessions from Spark workers. *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 Vista, since it too supports transfer learning. But unlike our work, *SparkDL* does not allow users to explore different CNN layers nor does it optimize query execution to improve reliability or efficiency. Thus, our work could augment *SparkDL*.

3 PRELIMINARIES AND OVERVIEW

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

Example Use Case (Based on [53]). Consider a data scientist at an online fashion retailer working on a product recommendation system (see Figure 1). She uses logistic regression to classify products as relevant or not for a user based on structured features such as price, brand, category, etc., and user behavior. There are also product images, which she thinks could help improve accuracy. Since building deep CNNs from scratch is too expensive for her, she uses the pre-trained deep CNN AlexNet [45] and uses the penultimate feature layer as the image representation. She also tries a few other layers and compares their accuracy. While this example is simplified, such use cases are growing across application domains, including online advertising (with ad images), nutrition and inventory management (with food/product images) [11], and healthcare (with tissue images) [33].

Comparing multiple CNN layers is crucial for effective transfer learning [17, 23, 32, 65]. As a sanity check experiment, we took the public *Foods* dataset [11] and built a classifier predict of a particular food item is a plant-based food or beverage or not. Using structured features alone (e.g., sugar and fat content), a well-tuned logistic regression model yields a test accuracy of 85.2%. Including image features from *fc6* layer of ResNet raises it to 88.3% (Appendix D).

3.1 Definitions and Data Model

We now introduce some definitions and notation to help us formalize the data model of partial CNN inference. These terms and notation will be used 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* FlattenOp *is a TensorOp whose output is a vector; given a tensor $t \in \mathbb{R}^{n_1 \times n_2 \times \ldots n_d}$, the output vector's length is $\prod_{i=1}^{d} n_i$.*

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

Definition 3.5. *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 3.6. 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 *$i \to j$ is the process of computing $f_j(\ldots f_i(t)\ldots)$, denoted $\hat{f}_{i \to j}$.*

Definition 3.8. Feature layer. *Given a CNN f, layer index i, and an image tensor t that is shape-compatible with layer f_i,* feature layer *l_i is the tensor $\hat{f}_i(t)$.*

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

3.2 Problem Statement and Assumptions

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

[1]For exposition, we focus on sequential (chain) CNNs, but it is straightforward to extend our definitions to DAG-structured CNNs such as DenseNet as well [39].

each of the $|L|$ feature vectors obtained by concatenating X with the respective feature layers obtained by partial CNN inference. More precisely, we can state the the workload using the following set of logical queries:

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

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

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

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

Step (2) performs partial CNN inference to materialize feature layer l and flattens it with g_l, a shape-compatible FlattenOp. Step (3) concatenates structured and image features using a key-key join. Step (4) trains M on the new multimodal feature vector. Pooling can be injected before g_l to reduce dimensionality for M [23]. The current dominant practice is to run the above queries as such, i.e., materialize feature layers *manually* and *independently* as flat files and transfer them; we call this approach *lazy materialization*. Apart from being cumbersome, such an approach is inefficient due to *redundant* partial CNN inference and/or runs the risk of system crashes due to poor memory management. Our goal is to resolve these issues. *Our approach is to elevate this workload to a declarative level, obviate manual feature transfer, automatically reuse partial CNN inference results, and optimize the system configuration and execution for better reliability and efficiency.*

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

3.3 System Architecture and API

We prototype VISTA as a library on top of two environments; Spark-TensorFlow [6, 16] and Ignite-TensorFlow [24]. Due to space constraints we explain only the Spark-based prototype architecture; the prototype on Ignite is similar. Figure 2 illustrates our system architecture. It has three main components: (1) a "declarative" API, (2) a roster of popular named

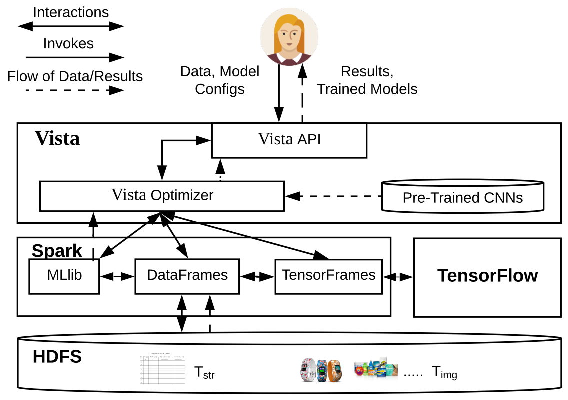

Figure 2: System architecture of the VISTA prototype on top of the Spark-TensorFlow combine. The prototype on Ignite-TenforFlow is similar and skipped for brevity.

```
/**
 * * * * * * * * * * * * * * * * * 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 3: VISTA API and sample usage showing values for the input parameters and invocation.

deep CNNs with named feature layers (we currently support AlexNet [45], VGG16 [60], and ResNet50 [35]), and (3) the VISTA optimizer. The declarative front-end API (see Figure 3) is implemented in Python; a user should specify several inputs, with three major groups of inputs. First are the system environment (memory, nodes and number of cores). Second are the deep CNN f and the number of feature layers $|L|$ (starting from the top most layer) to explore for transfer learning. Third is the downstream ML routine M, provided as a function pointer (we assume this routine handles the downstream model's artifacts). Fourth are the data tables T_{str} and T_{img} and statistics about the data. The result is a dictionary with the $|L|$ training errors.

Under the covers, VISTA invokes its optimizer (Section 4.3) to obtain a reliable and efficient combination of decisions for the logical execution plan (Section 4.2.1), key system configuration parameters (Section 4.2.2), and physical execution (Section 4.2.3). After configuring Spark accordingly, VISTA runs within the Spark Driver process to orchestrate the feature transfer task by invoking Spark's *DataFrame*, *TensorFrames* [16], and *MLlib* APIs. VISTA has user-defined functions for (partial) CNN inference, i.e., f, $\hat{f}_l$, g_l, and $\hat{f}_{i\rightarrow j}$ for the CNNs in its roster. These functions pre-specify the TF computational graphs to use. During query execution, Vista invokes the *DataFrame* and *TensorFrames* APIs with the appropriate user-defined functions injected based on the user inputs and optimizer decisions. Image and feature tensors are handled using our custom *TensorList* datatype. Overall, the user does not need to write any TF code. Finally, Vista uses MLlib to invoke the downstream ML algorithm on the joined multimodal feature vector and saves $|L|$ trained downstream models. *Overall, Vista frees users from having to manually handle TF code, save features as files, perform joins of RDDs, or tune Spark for such scalable feature transfer workloads.*

4 TRADE-OFFS AND OPTIMIZER

We analyze the abstract memory usage behavior of our workload and explain how it maps to Spark and Ignite memory models. We then use the analysis to explain the trade-off space for improving reliability and efficiency. Finally, we apply our analyses to design the Vista optimizer.

4.1 Memory Use Analysis of Workload

It is important to understand and optimize the memory use behavior of our workload, since mismanaged memory can cause frustrating system crashes and/or excessive disk spills/cache misses that raise runtimes in the distributed memory-based environment. 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 data system, we create an *abstract model of distributed memory apportioning* to help us explain the trade-offs in a generic manner. These trade-offs involve apportioning memory between intermediate data, CNN models, and working memory for UDFs, and they affect both reliability (avoiding crashes) and efficiency. We then highlight interesting new twists in our workload that can cause such crashes or inefficiency, if not handled carefully. Finally, for concreteness sake, we map our abstract memory model to two popular state-of-the-art distributed data processing systems, Spark and Ignite.

Abstract Memory Model. In distributed memory-based data processing 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. This is illustrated by Figure 4(A). For typical relational workloads, Execution Memory is further split into User Memory, which is used for UDF execution, and Core Memory, which is used for query processing. Common best practice guidelines recommend allocating most of System Memory to Storage Memory, while ensuring there is enough memory for Execution in order to reduce or avoid disk spills or cache misses [7, 13, 14]. OS Reserved Memory is generally set to a few GBs. But we note that such guidelines were designed primarily for relational workloads. Our workload requires rethinking memory apportioning due to interesting new twists caused by deep CNN models, (partial) CNN inference, feature layers, and the downstream ML task.

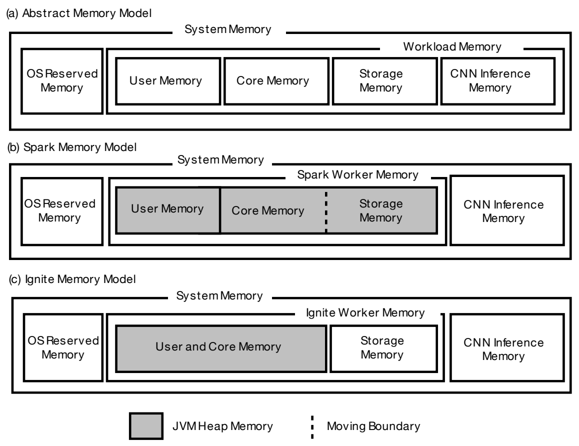

Figure 4: (A) Our abstract model of distributed memory apportioning. (B,C) How our model maps to Spark and Ignite.

First, the guideline of using most of System Memory for Storage and Execution Memory no longer holds. In both Spark-TF and Ignite-TF environments, CNN inference uses System Memory *outside* Storage and Execution Memory regions. The memory footprint of deep CNNs is non-trivial, e.g., AlexNet needs 2 GB. If we use multiple threads for parallelizing query execution in such parallel dataflow systems, each will spawn its own replica of the CNN, multiplying the footprint.

Second, many temporary objects are created in memory when reading serialized CNNs to initialize TF sessions and for buffers to read inputs and hold feature layers created by partial CNN 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. They could vary widely, and they could be massive. For example, 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 data scientists to handle manually.

Third, downstream ML algorithms also copy feature layers produced by TF into more amenable representations in order to process them. Thus, Storage Memory should accommodate these intermediate data copies. Finally, for the join between the table with the feature layers and T_{str}, Core Memory should accommodate temporary data structures created by system operations, e.g., the hash table on T_{str} for broadcast 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[2], 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 Spark needs more of the latter, it borrows automatically from the former by evicting cached data partitions using an LRU cache replacement policy. Conversely, if Spark needs to cache more data, it borrows from Execution Memory. But there is a maximum threshold fraction of Storage Memory (default 50%) that is immune to eviction. Worker threads in Spark run in isolation and do not have 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 in our workload listed earlier give rise to various, potentially unexpected, system crash scenarios due to memory errors, as well as inefficiency issues. Having to avoid these issues manually could frustrate data scientists and impede their ML exploration.

(1) CNN blowups. Human-readable file formats of CNNs often underestimate their in-memory footprints. Along with the replication of CNNs by multiple threads, CNN Inference Memory can be easily exhausted. If users do not account for such blowups when configuring the data processing system, and if the blowups 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 trade-offs precisely along three dimensions.

4.2 Dimensions of Trade-offs

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

4.2.1 Logical Execution Plan Trade-offs. The first step is to improve upon the lazy materialization approach (Section 3.2) to avoid computational redundancy and reduce memory pressure. To see why redundancy exists, consider a popular deep CNN AlexNet with the last two fully-connected layers *fc7 and fc8* tried for feature transfer ($L = \{fc7, fc8\}$). The *Lazy* plan, shown in Figure 5 (a), performs partial CNN inference for *fc7* (721 MFLOPS) independently of *fc8* (725 MFLOPS), incurring almost 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 note that this plan still has computational redundancy. The only way to remove redundancy is to break the independence of the $|L|$ queries and fuse them. This is a CNN-aware form of multi-query optimization [59]. Realizing this optimization requires new TensorOps for partial CNN inference, which we are able to handle because we do not treat CNN inference as a black box.

The first new plan we consider is the *Eager* plan, shown in Figure 5 (c). It materializes all feature layers of L *in one go* to avoid redundancy. The features are stored as a *TensorList* in an intermediate table and joined with T_{str}. M is then 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. Empirically, we find that

[2]Spark also leave out 300MB of memory from heap as a safety margin, but this detail does not affect the generic trade-offs we discuss.

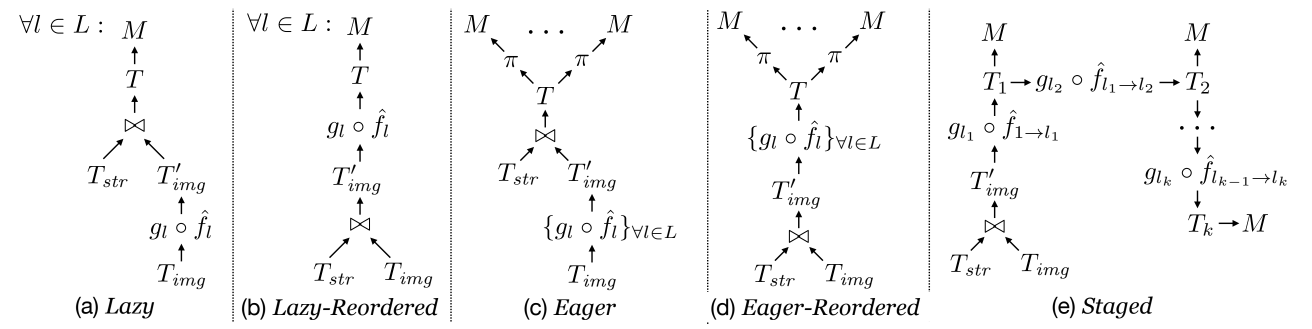
$\forall l \in L:$ M T $\bowtie$ T_{str} T'_{img} $g_l \circ \hat{f}_l$ T_{img}
(a) *Lazy*
$\forall l \in L:$ M T $g_l \circ \hat{f}_l$ T'_{img} $\bowtie$ T_{str} T_{img}
(b) *Lazy-Reordered*
M ... M π π T $\bowtie$ T_{str} T'_{img} $\{g_l \circ \hat{f}_l\}_{\forall l \in L}$ T_{img}
(c) *Eager*
M ... M π π T $\{g_l \circ \hat{f}_l\}_{\forall l \in L}$ T'_{img} $\bowtie$ T_{str} T_{img}
(d) *Eager-Reordered*
M $T_1 \rightarrow g_{l_2} \circ \hat{f}_{l_1 \rightarrow l_2} \rightarrow T_2$ M $g_{l_1} \circ \hat{f}_{1 \rightarrow l_1}$ T'_{img} $\bowtie$ T_{str} T_{img} ... $g_{l_k} \circ \hat{f}_{l_{k-1} \rightarrow l_k}$ $T_k \rightarrow M$
(e) *Staged*

Figure 5: Alternative logical query plans. Plan (a) is the *Lazy materialization* plan, the de facto practice today. Plan (b) reorders the join operator in plan (a). Plan (c) is the *Eager materialization* plan. Plan (d) reorders the join in plan (c). Plan (e) is our new *Staged materialization* plan. We define $k = |L|$.

CNN inference operations dominate overall runtime (85–99% of total); thus, join placement does not matter much for runtime although but it eases memory pressure. Nevertheless, *Eager* and *Eager-Reordered* still have high *memory pressure*, since they materialize all of L at once. Depending on the memory apportioning (Section 4.1), this could cause crashes or a lot of disk spills, which in turn raises runtimes.

To resolve the above issues, we create a logical execution plan we call *Staged* materialization, shown in Figure 5 (e). It splits partial CNN inference across the layers in L and invokes M on branches off the inference path. *Staged* avoids redundancy and has lower memory pressure, since feature materialization is staged out. Interestingly, *Eager* and *Eager-Reordered* are seldom much faster than *Staged* due to a peculiarity of deep CNNs. For the former to be much faster, the CNN must "quickly" (i.e., within a few layers and low MFLOPs) convert 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 [34]. In fact, almost no popular deep CNN model has such an architecture. This means *Staged* typically suffices from both the efficiency and reliability standpoints (we validate this in Section 5). Thus, unlike conventional optimizers that consider multiple logical plans, we use only the *Staged* plan in Vista.

4.2.2 System Configuration Trade-offs. Logical execution plans are generic and independent of the data system used. But as explained in Section 4.1, many system configuration parameters have a direct impact on reliability and efficiency. Thus, we need to understand and optimize the trade-offs of setting such parameters automatically. In particular, we need to set the degree of parallelism in a worker, data partition sizes, and memory apportioning.

Naively one might set the degree of parallelism to the number of cores on the node, allocate few GBs for User and Core Execution Memory, a majority of the remaining system memory for Storage Memory, and leave the number of partitions to the default value in the data system. Such naive settings can cause memory-related crashes or inefficiencies. But tuning these parameters to avoid such issues manually is tedious and non-trivial, since they are inter-dependent: a higher degree of parallelism increases the throughput for a worker but also raises the CNN models' footprint. In turn, this means reducing Execution and Storage Memory, which in turn means the number of partitions should be raised.

Reducing Storage Memory will cause more disk spills, especially for feature layers, and raise runtimes. Worse still, User Memory might also become too low, which could cause crashes during UDF execution. 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 worker nodes might get underutilized [3]. In turn, such underutilization of parallelism could raise runtimes, especially for the join and the downstream ML model in our workload. Finally, too low a number of data partitions can cause crashes, while too high a value leads to high overhead for processing too many data partitions. Overall, one needs to navigate such non-trivial systems trade-offs that are closely tied to the CNN model, the sizes of the layers being compared and the downstream ML model.

4.2.3 Physical Execution Trade-offs. The physical execution trade-offs are also largely determined by the specifics of the underlying data system, but two major physical execution decisions are usually commonly required regardless of the data system used.

The first decision is the physical join operator to use. The two main options for distributed joins are *shuffle-hash join* and *broadcast join*. In a shuffle-hash join, base tables are hashed on the join attribute and partitioned into "shuffle

[3]We note, however, that in the current Spark-TF and Ignite-TF environments, 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 throughput and reduce runtimes.

blocks." Each shuffle block is then send 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 a broadcast join, each worker is sent a copy of the smaller table on which it builds a local hash table and joins it with the outer table without any shuffles. If the smaller table fits in memory, broadcast join is typically faster due to lower communication and disk I/O overheads.

The second decision 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 or reduce disk spills or cache misses. The two main options are to store the data in *deserialized* format or in a more 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 need to estimate size of the intermediate data tables $|T_i|$ (for $i \in L$). This requires understanding the internal record format used by the underlying data system, which we account for in Vista (Appendix A).

We note that Spark supports both shuffle-hash join and broadcast join implementations, as well as both serialized and deserialized in-memory storage formats. In Ignite, data will be 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 the above dimensions of trade-offs automatically to improve system efficiency and reliability. The optimizer is based on our abstract memory model. Table 1 lists all the notation used in this subsection.

Optimizer Formalization and Simplification. The inputs for the optimizer are listed in Table 1(A). Table 1(B) lists the variables set by the optimizer. $|f|_{ser}$, $|f|_{mem}$, and $|f|_{mem\_gpu}$ are not input directly by the user; Vista has this knowledge of f in its roster. Similarly, $|M|$ is also not input directly by the user; Vista estimates it based on the specified M and the largest total number of features (based on L). For instance, for logistic regression, $|M|$ is proportional to $(|X| + \max_{l \in L} |g_l(\hat{f}_l(I))|)$. We define two quantities to capture peak intermediate data sizes and help our optimizer set memory parameters reliably:

Table 1: Notation for Section 4 and Algorithm 1.

Symbol	Description
(A) Inputs given/ascertained from workload instance	
$\lvert f\rvert_{ser}$	Serialized size of CNN model f
$\lvert f\rvert_{mem}$	In-memory footprint of CNN model f
$\lvert f\rvert_{mem\_gpu}$	GPU memory footprint of CNN model f
L	List of feature layer indices of f user wants to transfer
n_{nodes}	Number of worker nodes in cluster
mem_{sys}	Total system memory available in a worker node
mem_{GPU}	GPU memory if GPUs are available
cpu_{sys}	Number of cores available in a worker node
$\lvert T_{str}\rvert$	Size of the structured features table
$\lvert T_{img}\rvert$	Size of the table with images
$\lvert T_i\rvert$	Size of intermediate table T_i with feature layer $L[i]$ of f as per Figure 5(E); see Equation 15
$\lvert M\rvert$	User Memory footprint of downstream model
(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_1	Fudge factor for size blowup of storage data objects (default: 1.2)
α_2	Fudge factor for size blowup of binary feature vectors as JVM objects (default: 2)

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

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

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. 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. First, estimating τ is highly tedious, since it involves join costs, data loading costs, downstream model costs, etc. Second, and 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 [13, 15]. Thus, we cap *cpu* at $cpu_{max} = 8$. Third, given this cap, 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, these insights yield a much 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:

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

$$mem_{user} = \begin{cases} \text{(a) no shared memory:} \\ \quad cpu \times \max\{|f|_{ser} + \alpha_2 \times \lceil s_{single}/n_p \rceil, |M|\}, \\ \text{(b) shared memory:} \\ \quad \max\{|f|_{ser} + cpu \times \alpha_2 \times \lceil s_{single}/n_p \rceil, \\ \quad\quad cpu \times |M|\} \end{cases} \tag{10}$$

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

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

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

If GPUs are available:

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

Equation 9 caps *cpu* and leaves a CPU for the OS. Equation 10 captures User Memory needed 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 shown in Equation 10 (b). $cpu \times |f|_{mem}$ is the CNN Inference Memory needed for TF. Equation 11 constrains the total memory as per Figure 4. If there is access to GPUs, total GPU memory footprint $cpu \times |f|_{mem\_gpu}$ should be upper 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. With above observations, the algorithm is simple: linear search on *cpu* to satisfy all constraints. [4]Algorithm 1 presents it formally. If **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.

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 handles the trade-off space.

Datasets. We use two real-world datasets: *Foods* [11] and *Amazon* [36]. *Foods* has about 20,000 examples with 130 structured numeric features such as nutrition facts along with pairwise and ternary 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 list of categories, as well as a product image. The target represents the sales rank, which we binarize as a popular product or not. We pre-processed title strings to extract 100 numeric features (an "embedding") using the popular Doc2Vec procedure [49]. We convert the indicator vector for categories into

[4]We explain our algorithm only 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.

```
1:  procedure OptimizeFeatureTransfer:
2:      inputs: see Table 1(A)
3:      outputs: see Table 1(B)
4:      for $x = \min\{cpu_{sys}, cpu_{max}\} - 1$ to 1 do    ▷ Linear search
5:          $n_p \leftarrow$ NumPartitions($s_{single}, x, n_{nodes}$)
6:          $mem_{worker} \leftarrow mem_{sys} - mem_{os\_rsv} - x \times |f|_{mem}$
7:          $mem_{user} \leftarrow x \times \max\{|f|_{ser} + \alpha_2 \times \lceil s_{single}/n'_p \rceil, |M|\}$
8:          if $mem_{worker} - mem_{user} > mem_{core}$ then
9:              $cpu \leftarrow x$
10:             $mem_{storage} \leftarrow mem_{worker} - mem_{user} - mem_{core}$
11:             $join \leftarrow shuffle$
12:             if $|T_{str}| < b_{max}$ then
13:                 $join \leftarrow broadcast$
14:             $pers \leftarrow deserialized$
15:             if $mem_{storage} < s_{double}$ then
16:                 $pers \leftarrow serialized$
17:             return ($mem_{storage}, mem_{user}, cpu, n_p, join, pers$)
18:     throw Exception(No feasible solution)
19:
20: procedure NumPartitions($s_{single}, x, n_{nodes}$):
21:     $totalcores \leftarrow x \times n_{nodes}$
22:     return $\lceil \frac{s_{single}}{p_{max} \times totalcores} \rceil \times totalcores$
```

100 numeric features using PCA. All images are resized to 227 × 227 resolution, as required by most popular CNNs. All of our data pre-processing scripts and system code will be made available on our project web page. We hope our efforts help spur more research on this topic.

Workloads. We use three popular ImageNet-trained deep CNNs: AlexNet [45], VGG16 [60], and ResNet50 [35], obtained from [5, 10]. They complement each other in terms of model size and total MFLOPs [25]. We select the following interesting 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 [35]), with only the topmost layer being fully-connected. Following standard practices [17, 65], we apply max pooling on the convolutional feature layers to reduce their dimensionality before using them for M[5]. As for M, we run 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 [57]. Each node has 32 GB RAM, Intel Xeon @ 2.00GHz CPU with 8 cores, and 300 GB Seagate Constellation ST91000640NS HDDs. They run Ubuntu 16.04. For the Spark-TF environment, we use Spark v2.2.0 with *TensorFrames* v0.2.9 integrating it with TensorFlow v1.3.0 and for the Ignite-TF environment, we use Ignite v2.3.0 with TensorFlow v1.3.0. Spark runs in standalone mode. Each worker runs one executor. HDFS replication factor is three; input data is ingested to HDFS and read from there. Ignite is configured with native persistence enabled with each cluster node running a single worker. Each runtime reported is the average of three runs with 90% confidence intervals.

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

5.1 End-to-End Reliability and Efficiency

We compare VISTA with five baselines: three naive and two strong. *Lazy-1* (1 CPU per Executor), *Lazy-5* (5 CPU per Executor), and *Lazy-7* (7 CPUs per Executor) represent the current dominant practice of lazy materialization (Section 3.2). Spark is configured based on best practices [7, 13] (29 GB JVM heap, deserialized, shuffle join, 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 value of 1024. *Lazy-5 with Pre-mat* and *Eager* are strong baselines based on our analysis of the logical plan trade-offs (Section 4.2.1). In *Lazy-5 with Pre-mat*, the lowest feature layer (e.g., *conv5* for AlexNet) is materialized beforehand and used in place of raw images for all subsequent CNN inference; *Pre-mat* is time spent on the pre-materialization part. *Eager* is eager materialization plan explained in Section 4.2.1 (with 5 CPUs per Executor). For *Lazy-5 with Pre-mat* and *Eager*, we explicitly apportion CNN Inference memory. VISTA shows the plan picked by our optimizer, including for system configuration (Section 4.3). Note that *Lazy-5 with Pre-mat* and *Eager* actually require parts of the VISTA code base. Figure 6 presents the results.

We see that VISTA improves reliability and/or efficiency across the board. In the Spark-TF environment, *Lazy-5* and *Lazy-7* crash on both datasets for VGG16; *Eager* crashes on *Amazon* for VGG16 and ResNet50. In the Ignite-TF environment, *Lazy-7* crashes for all models on *Amazon*, while for ResNet50, *Lazy-7* on *Foods* and *Eager* on *Amazon* also crash. These crashes are due to memory pressures caused by CNN model blowups or User Memory blowups (Section 4.1). When *Eager* does not crash, its efficiency is comparable to VISTA, which validates our analysis in Section 4.2.1. *Lazy-5 with Pre-mat* does not crash, but its efficiency is comparable to *Lazy-5* and worse than VISTA. This is because the feature layers of AlexNet and ResNet are much larger than the raw images, which raises data I/O and join costs (Appendix C provides runtime breakdowns). 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 explored, the more redundancy there is and the faster VISTA will be. We also found that if GPUs

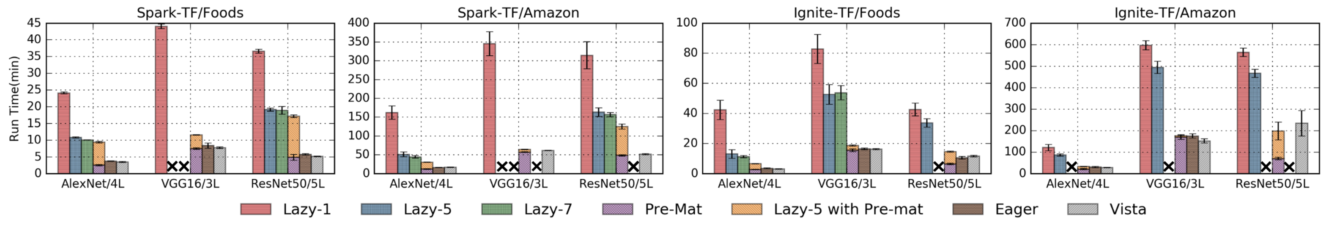

Figure 6: End-to-end reliability and efficiency. "×" indicates a system crash. Overall, Vista offers the best or near-best performance and never crashes, while the alternatives are much slower or crash in some cases.

are used for CNN inference, the overall trends are still the same even though CNN inference runtimes are significantly reduced; due to space constraints, we present the GPU results in Appendix E. Overall, Vista never crashes and offers the best (or near-best) efficiency on these workloads. This confirms the benefits of an automatic optimizer such as ours for improving reliability and efficiency, which could reduce both user frustration and costs.

5.2 Drill-Down Analysis of Trade-offs

We now analyze how Vista handles each of the three dimensions of trade-offs discussed in Section 4. We use the Spark-TF prototype of Vista, since it is faster than Ignite-TF. We use the less resource-intensive *Foods* dataset but alter it "semi-synthetically" for some experiments to analyze Vista performance in new operating regimes. In particular, when specified, we vary the data scale by replicating tuples (denoted, e.g., as "4X") or varying the number of structured features (with random values). For the sake of uniformity, unless specified otherwise, we use all 8 workers, fix *cpu* to 4, and fix Core Memory to be 60% of the JVM heap. We set the other parameters as per the Vista optimizer. The layers explored for each CNN are the same as before.

Logical Plan Decisions. We compare four combinations: *Eager* or *Staged* plan combined with inference *After Join* or *Before Join*. We vary both $|L|$ (by dropping successive lower layers) and data scale for AlexNet and ResNet. Figure 7 shows the results. We see that the runtime differences between all plans are insignificant for low data scales or low $|L|$ on both CNNs. But as $|L|$ or the data scale goes up, both *Eager* plans get much slower, especially for ResNet (Figure 7(B,D)), due to more disk spills for the massive intermediate table generated. Across the board, *After Join* plans are mostly comparable to their *Before Join* counterparts but marginally faster at larger scales. These results validate our choice of only using the *Staged*/*After Join* plan combination, which was plan (e) in Figure 7 in Section 4.2.1, in Vista.

Physical Plan Decisions. We compare four combinations: *Shuffle* or *Broadcast* join and *Serialized* or *Deserialized* 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*/*After Join*. Figure 8 shows the results. We see that all four plans are almost indistinguishable regardless of the data scale for ResNet (Figure 8(B)), except at the 8X scale, when the *Serialized* plans slightly outperform the *Deserialized* plans. For AlexNet, the *Broadcast* plans slightly outperform the *Shuffle* plans (Figure 8(A)). Figure 8(C) shows that this gap remains as $|X_{str}|$ increases but the *Broadcast* plans crash eventually. For ResNet, however, Figure 8(D) shows that both *Serialized* plans are slightly faster than their *Deserialized* counterparts, but the *Broadcast* plans still crash eventually. The gap between *Serialized* and *Deserialized* is more significant for ResNet than AlexNet, since at the 8X scale, its largest intermediate table requires disk spills. The Vista optimizer handles these trade-offs automatically.

System Configuration Decisions. We vary *cpu* and n_p, with the optimizer setting the memory parameters accordingly. The logical-physical plan combination is *Staged*/*After Join*/*Shuffle*/*Deserialized*. Figures 9(A,B) show the results for the three 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 the CNN Inference Memory requirement. The runtime decrease with *cpu* is, however, sublinear. 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 about 4 cores. As mentioned in Section 4.3, this is to be expected, since CNN inference dominates total runtime and TF always uses all cores regardless of *cpu* anyway. Appendix C provides the exact runtime breakdowns.

Figure 9(B) shows non-monotonic behaviors with n_p. At very low n_p, Spark crashes due to insufficient Core Memory for the join. As n_p goes up, runtimes go down, since Spark exploits more of the available parallelism (up to 32 usable cores). But eventually, runtimes rise again due to Spark overheads for handling too many tasks. In fact, when $n_p > 2000$, Spark compresses the task statuses sent to the master, which increases overhead substantially. The Vista optimizer sets n_p at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, which yield close to the fastest runtimes.

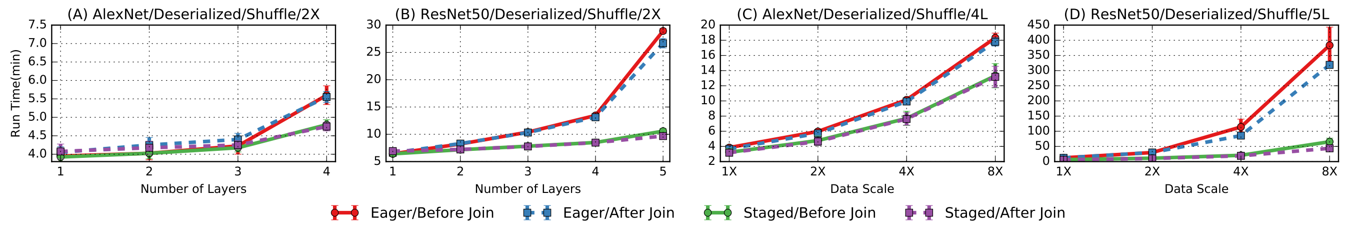

Figure 7: Runtime comparison of logical plan decisions for varying data scale and number of feature layers explored.

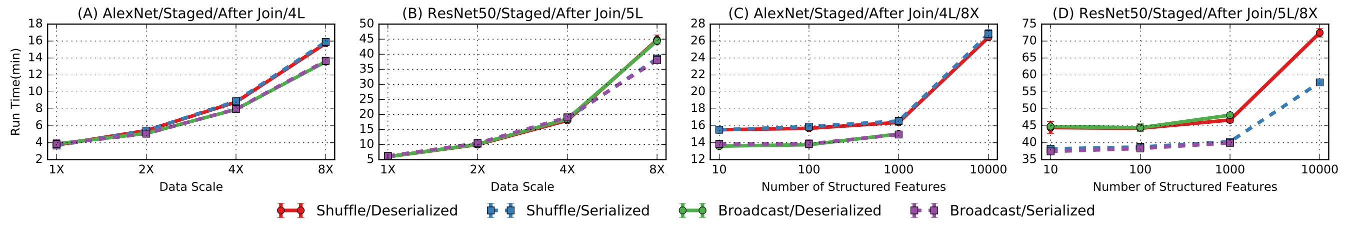

Figure 8: Runtime comparison of physical plan decisions for varying data scale and number of structured features.

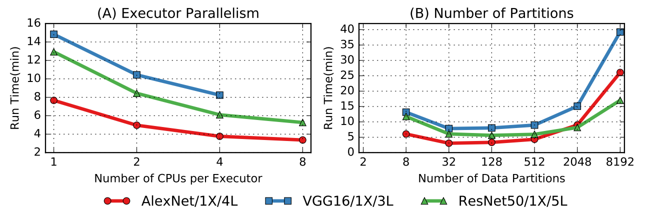

Figure 9: Varying system configuration parameters. Logical and physical plan choices are fixed to *Staged*, *After Join*, *Shuffle*, and *Deserialized*.

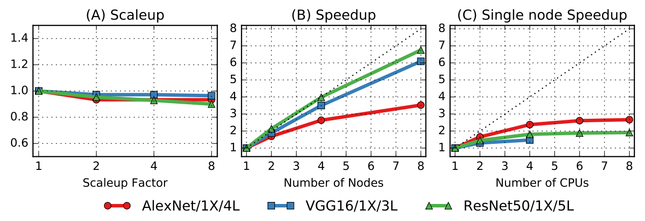

Figure 10: (A,B) Scaleup and speedup on cluster. (C) Speedup for varying *cpu* on one node with 0.25x data. Logical and physical plan choices are fixed to *Staged*, *After Join*, *Shuffle*, and *Deserialized*.

Scalability. Finally, we evaluate the speedup (strong scaling) and scaleup (weak 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 partial 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 [12]. 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. Appendix C provides more analysis of the speedups.

Accuracy. We also checked if the accuracy of the downstream ML model is improved when CNN features are added. We see accuracy lifts of about 3% overall, which is considered significant in ML practice. But no single feature layer of a given CNN dominates on accuracy, which validates the need for a tool like Vista to make it easier and faster to compare different feature layers. Since accuracy is orthogonal to this paper's focus, we discuss further details in Appendix D due to space constraints.

Summary of Experimental Results. Overall, ignoring the interconnected trade-offs of logical execution plan, system configuration, and physical execution plan often raises runtimes (even by 10x) or causes crashes. *Staged* inference significantly outperforms both *Lazy* (the current dominant practice) and *Eager* materialization at large scales. Pulling partial CNN inference above the join does not affect efficiency significantly but eases memory pressure. Proper system configuration for memory apportioning, data partitioning, and parallelism in a CNN- and feature layer-aware manner is crucial for reliability and efficiency. If the structured dataset is small, broadcast join marginally outperforms shuffle join but causes crashes at larger scales. Serialized disk spills are comparable to deserialized but marginally better in some cases. Overall, Vista manages and optimizes such complex systems trade-offs automatically, freeing data scientists to focus on their ML-related exploration.

5.3 Discussion and Limitations

TF is a powerful tool for building deep learning models but has poor support for data independence and structured data management, which forces users to manually manage data files, memory, distribution, etc. On the other hand, parallel dataflow systems and DBMSs offer better physical data independence. Thus, a marriage of these complementary frameworks will be beneficial for unified analytics over structured and unstructured data. But as our work 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 supports and optimizes large-scale feature transfer from deep CNNs for multimodal analytics combining structured data with images (one image per example). It currently supports a roster of popular CNNs for feature transfer and linear models for downstream ML and we did not consider secondary storage space a major concern, but 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 multimodal analytics tasks too, including image captioning [43]. Our focus is on systems for integrating images with structured features. A related but orthogonal line of work is "multimodal learning" in which deep neural networks (or other models) are trained from scratch on multimodal data [55, 61]. While feasible for some applications, this approach faces the same cost and data issues of training deep CNNs from scratch, which transfer learning mitigates.

Multimedia DBMSs. There is prior work in the database and multimedia literatures on DBMSs for "content-based" image retrieval (CBIR), video retrieval, and other queries over multimedia data [22, 41]. They relied on older handcrafted features such as SIFT and HOG [29, 52], not learned or hierarchical CNN features, although there is a resurgence of interest in CBIR with CNN features [64, 66]. Such systems are orthogonal to our work, since we focus on feature transfer with deep CNNs for multimodal analytics, not CBIR or multimedia queries. One could integrate Vista with multimedia DBMSs. NoScope is a system to quickly detect objects in video streams using cascades of CNNs [42]. Vista is orthogonal, since it focuses on feature transfer, not cascades.

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, [26, 37] studied the problem of predicate migration for optimizing complex relational queries with joins and UDF-based predicates. Also related is [18], which studied "semantic" optimization of queries with predicates based on data mining classifiers. Unlike such works on queries with UDFs in the `WHERE` clause, our work can be viewed as optimizing UDFs expressed 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 [59]. Vista is the first system to apply the general idea of MQO to complex CNN feature transfer workloads by formalizing partial CNN inference operations as first-class citizens for query processing and optimization. Vista can also be viewed as a model selection management system [47] that optimizes for CNN-based feature engineering. In doing so, our work expands a recent line of work on materialization optimizations for feature selection in linear models [44, 69] and integrating ML with relational joins [27, 46, 48, 58]. Finally, there is much prior work on auto-tuning system configuration for relational and MapReduce workloads (e.g., [38, 63]). Our work is inspired by those, but we focus specifically on the large-scale CNN feature transfer workload.

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 TensorFlow and parallel dataflow systems to support and optimize a key emerging workload in this context: feature transfer from deep CNNs for multimodal analytics. By enabling more declarative specification and by formalizing partial CNN inference, Vista automates much of the data management-oriented complexity of this workload, thus improving system reliability and efficiency, which in turn can reduce resource costs and potentially improve data scientist productivity. As for future work, we plan to support more general forms of CNNs and downstream ML tasks, as well as the interpretability of such models in data analytics.

REFERENCES

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

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

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

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

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

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

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

[8] Integrating ml/dl frameworks with spark. https://lists.apache.org/list.html?dev@spark.apache.org. Accessed January 31, 2018.

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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.

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

[23] 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.

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

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

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

[27] Chen, L., et al. Towards linear algebra over normalized data. *Proc. VLDB Endow. 10*, 11 (Aug. 2017), 1214–1225.

[28] Chou, H.-T., and DeWitt, D. J. An evaluation of buffer management strategies for relational database systems. *Algorithmica 1*, 1-4 (1986), 311–336.

[29] 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.

[30] 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.

[31] Deng, J., et al. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (2009), IEEE, pp. 248–255.

[32] 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.

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

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

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

[36] 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.

[37] 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.

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

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

[40] 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.

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

[42] Kang, D., et al. Optimizing deep cnn-based queries over video streams at scale. *CoRR abs/1703.02529* (2017).

[43] 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.

[44] Konda, P., et al. Feature selection in enterprise analytics: A demonstration using an r-based data analytics system. *Proc. VLDB Endow. 6*, 12 (Aug. 2013), 1306–1309.

[45] 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.

[46] Kumar, A., et al. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, ACM, pp. 1969–1984.

[47] Kumar, A., et al. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record 44*, 4 (2016), 17–22.

[48] Kunft, A., et al. Bridging the gap: Towards optimization across linear and relational algebra. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond* (2016), BeyondMR '16, ACM, pp. 1:1–1:4.

[49] 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.

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

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

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

[53] 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.

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

[55] 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.

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

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

[58] Schleich, M., et al. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, ACM, pp. 3–18.

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

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

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

[62] 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.

[63] 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.

[64] Wan, J., et al. Deep learning for content-based image retrieval: A comprehensive study. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), ACM, pp. 157–166.

[65] 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.

[66] Yue-Hei Ng, J., et al. Exploiting local features from deep networks for image retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2015), pp. 53–61.

[67] 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.

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

[69] Zhang, C., et al. Materialization optimizations for feature selection workloads. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14, pp. 265–276.

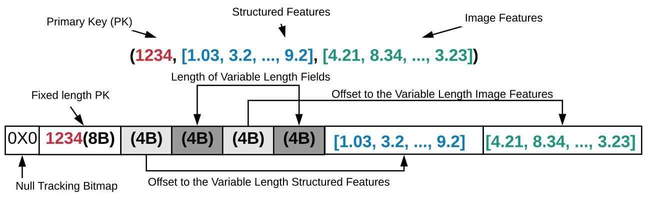

Figure 11: Spark's internal record storage format

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 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 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 $|T_i|$ with feature layer $l = L[i]$, we have:

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

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 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.

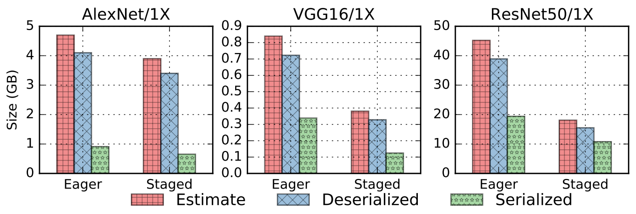

Figure 12: Size of largest intermediate table.

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)			
	1st	2nd	4th	5th
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 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 5.2.

For AlexNet and VGG16 when materializing 4th, 2nd, and 1st 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 5th layer features to the materialization time of 4th layer features. This can be attributed to the high disk IO overhead of writing out 5th layer image features which are ~3 times larger than that of 4th layer (see Figure. 13 (C)). Therefore, for ResNet50 starting

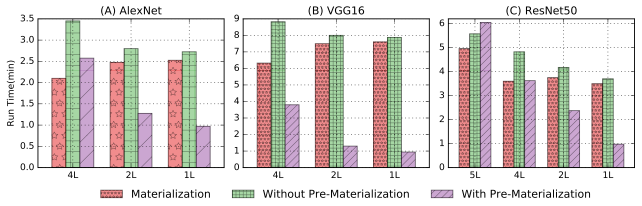

Figure 13: Runtimes comparison for using pre-materialized features from a base layer

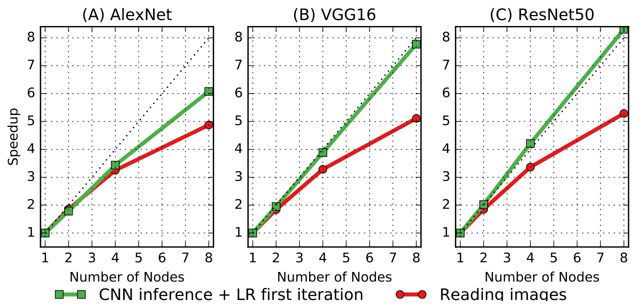

Figure 14: Drill-down analysis of Speedup Curves

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 [12]. 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.

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 14). 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.

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

Table 4: Accuracy lifts obtained by incorporating HOG descriptors and CNN features for logistic regression model with $||\ell||_1$ regularization.

	Structured Only	Structured + HOG	Structured + CNN
Foods	85.2	86.5	88.3
Amazon	65.4	66.3	68.4

D ACCURACY

For both Foods and a sample of Amazon (20,000 records) datasets we evaluate the downstream logistic regression model accuracy with (1) only using structured features, (2) structured features combined with "Histogram of Oriented Gradients (HOG)" [30] 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[6] 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.

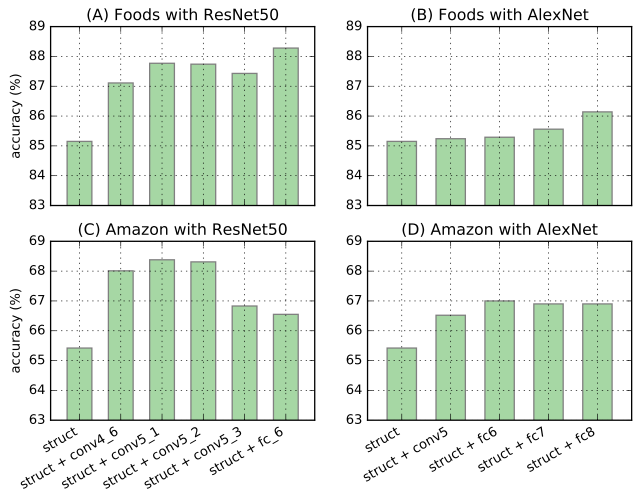

Figure 15: Accuracy lifts obtained by incorporating image features from different layers of CNN models for logistic regression model with $||\ell||_1$ regularization

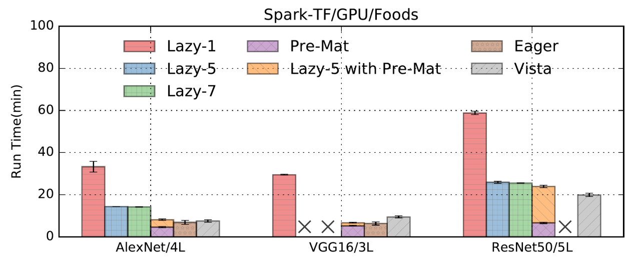

Figure 16: End-to-end reliability and efficiency on GPU. "×" indicates a system crash.

[6]Spark TensorFrames library was modified by adding TensorFlow GPU dependencies to enable GPU support