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

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

ABSTRACT

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

1. INTRODUCTION

Deep convolutional neural networks (CNNs) have revolutionized computer vision, yielding state-of-the-art accuracy for many image understanding tasks [46]. The main technical reason for their success is how they extract a hierarchy of relevant parametrized features from raw images, with the parameters learned automatically during training [34]. Each layer of a deep CNN learns a different level of abstraction in terms of what the features capture, e.g., low-level edges and patterns in the lowest layers all the way to abstract object shapes in the highest layers. This remarkable ability of deep CNNs is illustrated in Figure 1.

The success of deep CNNs presents an exciting opportunity to holistically integrate image data into traditional data analytics applications in the enterprise, Web, healthcare, and other domains that have hitherto relied mainly on structured data features but had auxiliary images that were not exploited. For instance, product recommendation systems such as Amazon are powered by ML algorithms that relied mainly on structured data features such as price, vendor, purchase history, etc. Such applications are increasingly using deep CNNs to exploit product images by extracting visually-relevant features to help improve ML accuracy, especially for products such as clothing and footwear [49]. Indeed, such deep CNN-based feature extraction already powers visual search and analytics at some Web companies [39]. Numerous other applications could also benefit from such "multimodal" analytics, including inventory management, healthcare, and online advertising [21].

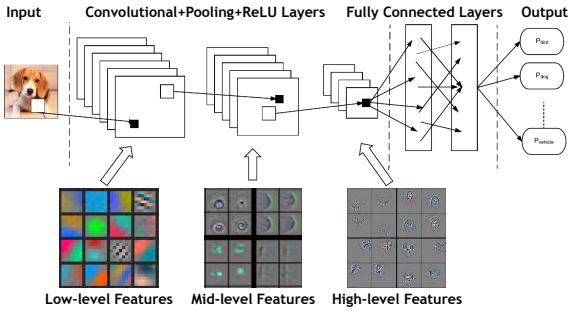


Figure 1: Simplified illustration of a typical deep CNN and its hierarchy of learned features (based on [68]).

Since training deep CNNs from scratch is expensive in terms of both resource costs (e.g., one might need many GPUs [2]) and the number of labeled examples needed, an increasingly popular paradigm to handle image data is *transfer learning* [53]. Essentially, one uses a pre-trained deep CNN, e.g., ImageNet-trained AlexNet [31, 43] 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 now operate on these image features along with the structured features, say, the popular logistic regression model or even "shallow" neural networks. Thus, such *feature transfer* helps reduce costs dramatically for using deep CNNs. Indeed, this paradigm is responsible for many high-profile successes of CNNs, including for detecting cancer [33] and diabetic retinopathy [61], facial analyses [19], and for product recommendations and search [39, 50].

Alas, feature transfer creates a new bottleneck for data scientists in practice: it is impossible to say in general which layer of a CNN will yield the best accuracy for the downstream ML task [26]. The common guideline is to extract and compare multiple layers of CNN features [26, 65]. This is a *model selection* process that combines CNN features and structured data [44]. Perhaps surprisingly, the current dominant approach to handling feature transfer at scale is for data scientists to manually *materialize* each CNN layer from scratch as flat files using tools such as TensorFlow [23], load such data into a scalable data analytics system for downstream ML tasks, say, using Spark and MLlib [51], which

is increasingly popular among enterprises [3]. Apart from reducing the productivity of data scientists, such manual management of feature transfer workloads leads to wasted opportunities to reuse and optimize computations, which raises runtimes and in turn, costs, especially in the cloud.

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

At first blush, feature transfer might seem straightforward: *Why not materialize and cache all layers of interest in one go and use a layer as needed?* While this reduces runtimes, as observed above, it increases *memory pressure*, since CNN features are often orders of magnitude larger than the input (e.g., one of ResNet50's layers is 784kB, while the input is 14kB [35]). Such data blowup lead to non-trivial systems trade-offs for handling memory usage at scale. In fact, performed naively, it could cause system crashes, which would frustrate data scientists and raise costs by forcing them to manually tweak the system or use needlessly more expensive machines. Also, caching unused layers can cause needless *disk spills*, which raises runtimes further. Thus, overall, large-scale feature transfer is technically challenging due to two key systems-oriented concerns: *reliability* (avoiding system crashes) and *efficiency* (reducing runtimes).

Resolving the above challenges requires navigating complex materialization trade-offs involving memory usage, feature storage, and execution runtimes. Since such trade-offs might be too low-level for most ML-oriented data scientists, we design a novel "declarative" data system to handle them and let users to focus on *what* layers they want to explore rather than *how* to run the workload. We prototype our system, named Vista, in the popular integrated Spark-TensorFlow environment [5,16] to leverage these systems for orthogonal benefits such as scalability, fault tolerance, and efficient CNN inference implementation.

We formalize CNN inference operations and perform a comprehensive analysis of the memory usage behavior of this workload on the Spark-TensorFlow combine. We then use our analysis to delineate three dimensions of systems trade-offs. First, we study novel *logical execution plan* choices to avoid redundant inference and also reduce memory pressure. Second, we study the trade-offs of key *system configuration* parameters, in particular, number of cores used by Spark, memory region sizes, and data partitioning. While best practice guidelines exist for such Spark parameters [1, 6], we show how novel twists in the feature transfer workload necessitate deviating from such guidelines. Third, we study the trade-offs of two *physical execution plan* choices, viz., join operator selection and spilled data serialization.

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

- To the best of our knowledge, this is the first paper to formalize and study the materialization trade-offs of the emerging workload of large-scale feature transfer from deep CNNs for multimodal analytics over image and structured data from a systems standpoint.
- Focusing on the popular Spark-TensorFlow environment, we delineate the trade-offs along the three dimensions of logical execution decisions, system configuration, and physical execution decisions. We introduce novel CNN-aware faster execution plans and explain how to optimize their memory management to help avoid system crashes and reduce runtimes.
- We devise a novel optimizer to handle such trade-offs automatically and build a system (named Vista) to enable data scientists to focus on their ML exploration instead of being bogged down by systems issues.
- We present an empirical evaluation of the reliability and efficiency of Vista using real-world datasets and CNNs and also analyze its handling of the trade-off space. Vista is able to catch and avoid many crash scenarios, while also reducing runtimes by up to 73%.

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

2. BACKGROUND

We now provide some relevant technical background and discuss other related work from both the machine learning/vision literature and the data systems literature.

Deep CNNs. CNNs are a type of neural networks specialized for image data [34, 46]. They exploit spatial locality of information in image pixels to construct a hierarchy of parametric feature extractors and transformers, which consist of a few types of layers: *convolutions*, which use image filters from computer 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 just a multi-layer perceptron. A "deep" CNN just stacks such layers many times over. All parameters are trained end-to-end using backpropagation [47]. This learning-based approach to feature engineering within CNNs enables them 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 [30,48]. In fact, deep CNNs have won numerous benchmark competitions in computer vision in the last few years [7, 55]. Popular deep CNN models from this recent line of work include AlexNet [43], VGG [58], Inception [60], and ResNet [35]. While our work is orthogonal

to how CNNs are designed, we note that deep CNNs incur a massive cost: they often require many GPUs for reasonable training times [2], and they need huge labeled datasets and "black magic" hyper-parameter tuning to avoid overfitting [34].

Transfer Learning with CNNs. Transfer learning is a popular paradigm to mitigate the above cost and data issues of training deep CNNs from scratch [53]. Essentially, one uses a pre-trained deep CNN, say one trained on ImageNet and downloaded from a "model zoo" [4, 9], removes its last few layers, and uses it as an image feature extractor. This "transfers" knowledge about images learned by the source CNN to new target prediction task that can use cheaper ML models. If the same CNN architecture is used but only the last few layers are retrained, it is called "fine tuning," but one could also use a different, more interpretable, ML model such as logistic regression instead. Such transfer learning with CNNs helped achieve breakthroughs in detecting cancer [33], diabetic retinopathy [61], face recognition-based analyses [19], and multimodal recommendation algorithms that combine images and structured data [50]. Recent works has also shown that such generic CNN features beat prior hand-designed features for many image analytics tasks [27, 32, 57, 57, 63, 64]. However, no single CNN layer is universally best for transfer learning; the only guideline is that the "more similar" the target task is to ImageNet, the better higher layer features are likely to be [17, 26, 32, 65]. Also, lower layer features are typically much larger; so, some form of feature selection such as extra pooling is typically performed [26]. Thus, data scientists have to try at least a few different layers of features from a deep CNN for best results with transfer learning [26, 65].

Spark and TensorFlow. Spark is a popular distributed memory-oriented and fault-tolerant data analytics system [1, 67]. At its core is the Resilient Distributed Dataset (RDD) abstraction, an immutable collection of key-value pairs that supports numerous dataflow operations, including relational operations and MapReduce. Queries compose such operations, and they are evaluated lazily. Spark uses HDFS for storage. It allows explicit caching of RDDs in distributed memory and supports disk spills during query processing. SparkSQL builds upon the RDD API to offer higher-level relational APIs–*DataFrame* and *DataSet*–and performs RDBMS-style query optimizations [25]. Spark guidelines now recommend the use of *DataFrame* or *DataSet* instead of RDDs [25]. MLlib (and SparkML) is a library that offers popular ML algorithms over Spark; its use for ML over structured data is growing, especially in enterprises [3, 8].

TensorFlow (TF) is a framework for expressing ML algorithms, especially complex neural network architectures (including deep CNNs) [22, 23]. Models in TF are specified as a "computational graph," with nodes representing operations over "tensors" (multi-dimensional arrays) and edges representing data flow. To execute a graph, one selects a node to run after giving all its input data. By separating these two stages, TF uses lazy evaluation to compile the full graph and apply some optimizations. TF is under active development and has a rapidly growing user base among deep learning researchers and engineers, especially for image and text data [11]. *TensorFrames* and *SparkDL* are APIs that integrate Spark and TF [15, 16]. They enable the use of TF within Spark by spawning TF processes from Spark workers. *TensorFrames* lets users process a *DataFrame* using TF code, while *SparkDL* offers pipelines to integrate deep neural networks into Spark queries and distribute hyper-parameter tuning. *SparkDL* is the most closely related work to Vista, since it too supports transfer learning. But unlike our work, *SparkDL* does not allow users to explore different CNN layers nor does it optimize query execution to improve reliability or efficiency. Thus, Vista could augment *SparkDL*.

3. PRELIMINARIES AND OVERVIEW

We start with a concrete example use case and then introduce some definitions and formalize our data model. We then state the problem we study, explain our assumptions, and give an overview of Vista.

Example Use Case (Inspired by [50]). Consider a data scientist at an online fashion retailer working on a product recommendation system. She uses logistic regression to classify products as relevant or not for a user based on structured features such as price, brand, category, etc., and user behavior. There are also product images, which she thinks could help improve accuracy. Since building deep CNNs from scratch is too expensive for her, she uses the pre-trained deep CNN AlexNet [43] to read off the penultimate feature layer as image features. She also tries a few other layers and compares their accuracy. While this example is simplified, such use cases are growing across application domains, including online advertising (with ad images) [21], nutrition and inventory management (with food/product images) [10], and healthcare (with tissue images) [33].

Comparing multiple CNN layers is crucial for effective transfer learning [17, 26, 32, 65]. As a sanity check experiment, we took the public *Foods* dataset [10] and built an ML classifier to predict nutrition level of a food item. Using structured features (e.g., sugar and fat content) alone, a well-tuned logistic regression model yields a test accuracy of 87.6%. Including image features from layer *fc6* of AlexNet lowers it to 86.8%, while *fc8* raises it to 90%!

3.1 Definitions and Data Model

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

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

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

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

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

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

[1] This definition is the same as in TensorFlow [23].

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

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

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

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

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

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

Note that all major layers in a CNN model–convolutional, pooling, non-linearity, and fully connected–are TensorOps. With the above definitions, we observe 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 automate memory management for, and optimize the execution of, feature transfer workloads, which we define next.

3.2 Problem Statement and Assumptions

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

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

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

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

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

Basically, step (2) performs partial CNN inference to get 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 the downstream ML model on the new multimodal feature vector. Pooling is sometimes injected before g_l to reduce dimensionality for M [26]. Perhaps surprisingly, the predominant practice today is to run the above queries as such, i.e., materialize each feature layer *manually* and *independently* as flat files and use them for transfer learning. Apart from being cumbersome, such a manual approach is inefficient due to redundant partial CNN inference computations 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 management of features, automatically reuse partial CNN inference results, and optimize the system configuration and execution for better reliability and efficiency.*

[2]For exposition sake, we focus on sequential (chain) CNNs, but it is straightforward to extend our definitions to DAG-structured CNNs such as DenseNet as well [38].

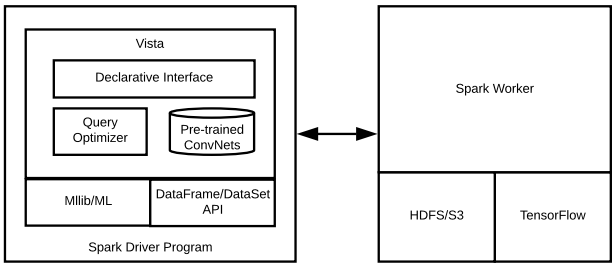


Figure 2: High-level architecture of VISTA on top of the Spark-TensorFlow combine.

We make a few simplifying assumptions in this paper for tractability sake. First, we assume that f is from a roster of well-known models such as AlexNet, VGG, and ResNet supported by VISTA. This is a reasonable start, since almost all recent transfer learning applications used only such well-known CNNs downloaded from the so-called "model zoos" [4, 9]. We leave support for arbitrary CNN architectures to future work. Second, we focus on using logistic regression for M, specifically, with MLlib (using L-BFGS). But this choice is largely orthogonal to our focus; it lets us understand CNN feature materialization trade-offs in depth. It is interesting future work to support more ML models for M, e.g., multi-layer perceptrons. Finally, we assume that secondary storage is plentiful, since it is much cheaper than memory or compute resources. This lets us focus on crucial memory-related issues of this workload, since Spark is distributed memory-oriented and handles disk spills.

3.3 System Architecture and API

We prototype VISTA as a library on top of the Spark-TensorFlow combine [5,16]. Figure 2 illustrates our system's architecture. It has four main components: (1) a "declarative" API, (2) a roster of popular named deep CNNs (we currently support *AlexNet* [43], *VGG16* [58], *ResNet50* [35]) with feature layers named as per their conventions, (3) the VISTA optimizer, and (4) a set of interfaces to talk to the other parts of Spark and TensorFlow (TF). The front-end of VISTA is implemented in Python; a user should specifies four inputs in our declarative API. First is the system environment (number of cores, system memory, and number of nodes). Second is the deep CNN f and the feature layers L to use for transfer learning. Third are the data tables T_{str} and T_{img}. Fourth is the downstream ML routine (with all its parameters)–currently MLlib's logistic regression.

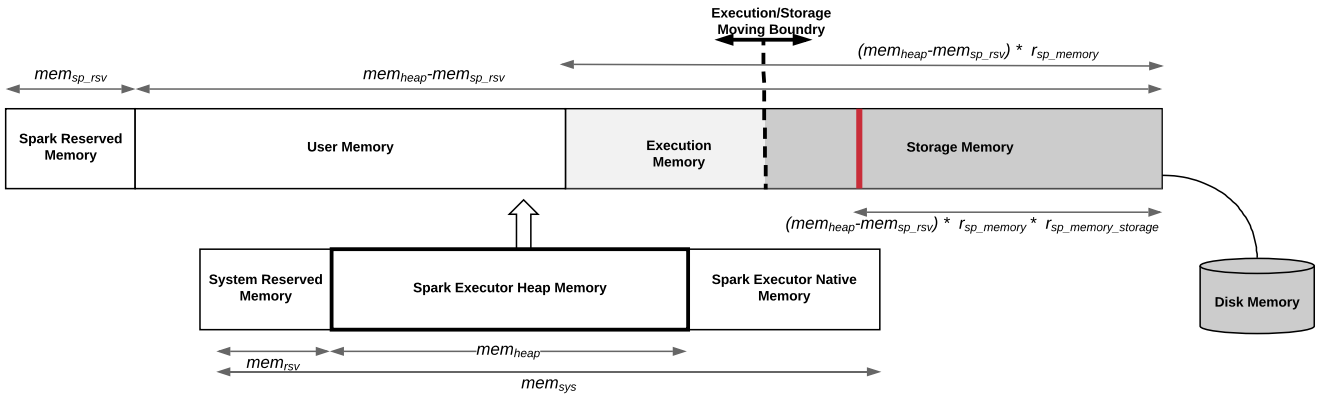


Figure 3: Memory management scheme of Spark and its interactions with TensorFlow and Vista.

Under the covers, Vista uses the above inputs and invokes its optimizer (Section 4.3) to obtain a reliable and efficient combination of decisions for the logical execution plan (Section 4.2.1), key system configuration parameters (Section 4.2.2), and physical execution (Section 4.2.3). After configuring Spark accordingly, Vista runs within the Spark Driver process to orchestrate the feature transfer task by issuing a series of queries in Spark's *DataFrame* (or *DataSet*) API [25]. Vista uses the *TensorFrames* API [16] to invoke TF during query execution to execute our user-defined functions for partial CNN inference and to handle image and feature tensors using our user-defined TensorList datatype. Vista specifies the computational graphs to be used by TF based on the user's inputs. Finally, Vista invokes MLlib in the manner determined by our optimizer and returns all trained downstream models. *Overall,* Vista *frees users from having to manually handle TF code, large feature files, RDD joins, or Spark tuning for such feature transfer tasks.*

4. TRADE-OFFS AND OPTIMIZER

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

4.1 Memory Analysis of Workload

It is crucial to understand and optimize the memory usage of large-scale feature transfer in the Spark-TF environment, since mismanaged memory can cause frustrating system crashes, while excessive disk spills waste runtime. For simplicity sake, we assume each worker node runs a single Spark Executor, which is a JVM process. We first introduce the various memory regions of Spark based on [14]. We then discuss how our workload presents interesting new twists, which lead to system crashes or inefficiency, if not handled carefully. Figure 3 illustrates our analysis.

Overview of Spark's Memory Regions. A worker node's memory is split into two high-level regions: Reserved Memory for OS and other processes and Spark Executor Memory, which in turn is split into two sub-regions: Spark Executor Heap Memory (the maximum JVM heap size) and Spark Executor Native Memory. In a typical relational workload on Spark, memory use is dominated by the JVM heap, while the System Reserved Memory is only a few GBs (say, 2–4 GB). The user has to specify a maximum JVM heap size (and the number of cores) for an Executor. Spark best practice guidelines recommend giving as much memory as possible to the JVM heap to reduce disk spills [6, 13, 14].[3]

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

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

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

First, the guideline of allocating most of the system memory to Spark Executor Heap Memory no longer holds. In the Spark-TF combine, CNN inference uses Spark Executor Native Memory *outside* the Java process. The memory footprint of deep CNN models is non-trivial (e.g., AlexNet

[3]But if heap memory becomes extremely large, the JVM garbage collection overhead might be high. In such cases, multiple Executors per worker are recommended [6]. It is straightforward to extend Vista to such a setting.

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

AlexNet [43]			**VGG** (16 layer version) [58]			**ResNet** (50 layer version) [35]		
Layer	**Output Shape**	**MFLOPs**	**Layer**	**Output Shape**	**MFLOPs**	**Layer**	**Output Shape**	**MFLOPs**
image	[227,227,3]		image	[227,227,3]		image	[227,227,3]	
conv1	[55,55,96]	105	conv1_x	$[224, 224, 64] \times 2$	1943	conv1	[112, 112, 64]	236
conv2	[27,27,256]	224	conv2_x	$[112, 112, 128] \times 2$	2777	conv2_x	$[56, 56, 256] \times 3$	1354
conv3	[13,13,384]	150	conv3_x	$[56, 56, 256] \times 3$	4626	conv3_x	$[28, 28, 512] \times 4$	1910
conv4	[13,13,384]	112	conv4_x	$[28, 28, 512] \times 3$	4626	conv4_x	$[14, 14, 1024] \times 6$	2780
conv5	[13,13,256]	75	conv5_x	$[14, 14, 512] \times 3$	4626	conv5_x	$[28, 28, 512] \times 3$	1470
fc6	[4096]	38	fc6	[4096]	102	fc6	[1000]	4
fc7	[4096]	17	fc7	[4096]	17			
fc8	[1000]	4	fc8	[1000]	4			

needed 2 GB). Worse still, if the Executor uses multiple threads, each will spawn its own TF session with a replica of the CNN! Second, many temporary objects are created for reading serialized CNNs to initialize TF sessions and for buffers to read inputs and hold feature layers created by (partial) CNN inference. All of these go under User Memory; Spark has almost no guidelines for this region–it is entirely up to the user to ensure this memory is enough! But this region's memory demand depends on the number of examples, the CNN model, and the layers of interest. It varies widely, and it could be massive. To illustrate this point, Table 1 lists the feature layer sizes for a few popular CNNs. We see that *fc6* of AlexNet is of length 4096, but *conv5* of ResNet is over 400,000. Performing such memory footprint calculations could be tedious for data scientists.

Third, Spark copies feature layers obtained with TF into RDDs to let the downstream ML model (in MLlib) process them. Thus, Storage Memory should accommodate the new RDD(s). Finally, for the join between the table with the feature layers and T_{str}, Execution Memory should accommodate temporary data structures created by Spark's operations, e.g., the hash table on T_{str} for broadcast join.

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

(1) CNN blowups. Human-readable file formats of CNNs often underestimate their in-memory footprints. Along with the replication of CNNs by multiple threads, Spark Executor Native Memory can be easily exhausted. Users must account for this when configuring the JVM heap size. If such blowups exceed available memory, the OS will kill the Executor.

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

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

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

Overall, we see that several execution and system configuration considerations affect system reliability and efficiency. Next, we delineate these systems trade-offs precisely along three dimensions.

4.2 Dimensions of Trade-offs

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

4.2.1 Logical Execution Plan Trade-offs

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

The first new plan we create is Plan C, "*Bulk Inference*." It materializes all feature layers of L *in one go* to avoid redundancy. The features are stored as a TensorList in an intermediate table and joined with T_{str}. M is then run on each feature layer (concatenated with X) projected from the TensorList. Plan D is the variant in which the join is pulled down. Empirically, we find that CNN inference operations dominate runtime (85–99% of total time) and

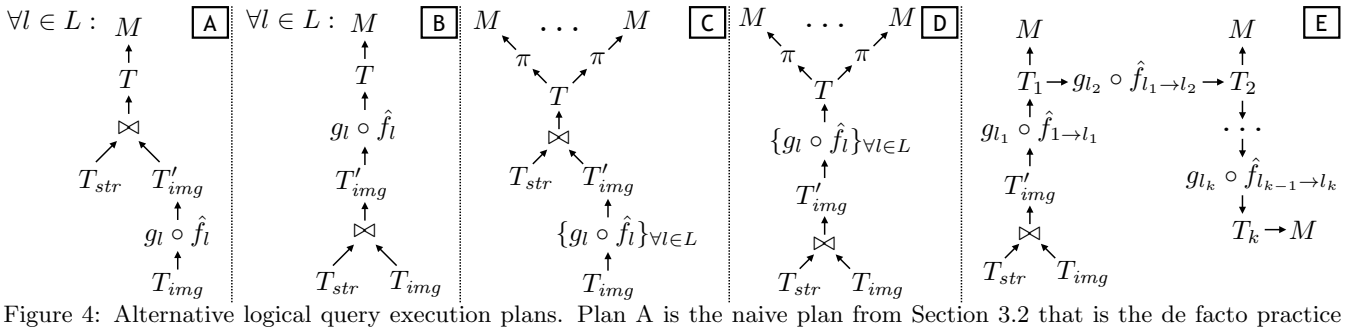


Figure 4: Alternative logical query execution plans. Plan A is the naive plan from Section 3.2 that is the de facto practice today. Plan C and D are the new "Bulk Inference" plans, while Plan E is a new "Staged Inference" plan. We define $k = |L|$.

thus, join placement does not matter much for runtime. But it helps ease memory pressure. Still, Plans C and D have high memory pressure, since they materialize all of L at once. This could cause system crashes, if the JVM heap is not large enough; even if we avoid crashes, these plans could face a lot of disk spills, which increase runtime.

To resolve the above issues, we create a novel plan, "*Staged Inference*," Plan E in Figure 4. It splits partial CNN inference across the layers in L and invokes M on branches off of the inference path. Plan E avoids redundancy and has lower memory pressure, since feature materialization is staged out. Interestingly, Plans C/D are seldom much faster than Plan E due to a peculiarity of deep CNNs. For Plans C/D to be much faster, the CNN must "quickly" (i.e., with few layers and low MFLOPs) convert the image to small feature tensors. But such an architecture is unlikely to yield high accuracy, since it loses too much information too soon [34]. In fact, almost no popular CNN model has such an architecture. This means Plan E typically suffices from both the reliability and efficiency standpoints (we validate this in Section 5). Thus, unlike conventional optimizers that consider multiple logical plans, VISTA uses only Plan E.

4.2.2 System Configuration Trade-offs

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

4.2.3 Physical Execution Trade-offs

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

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

4.3 The Optimizer

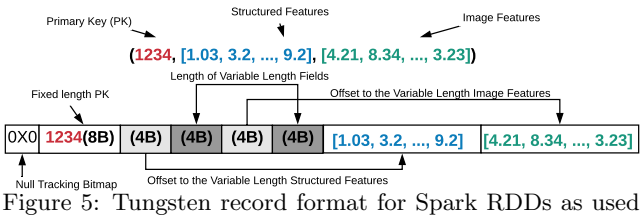


Figure 5: Tungsten record format for Spark RDDs as used in Vista for concatenated structured and image features.

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

Intermediate Data Sizes. As explained in Section 4.2.1, Vista only uses Plan E. But for system configuration and physical plan decision, Vista needs the sizes of the intermediate tables, $T_l, \forall l \in L$ in Figure 4(E), in the Tungsten record format. We estimate these automatically based on Vista's knowledge of f. For simplicity, assume ID is a long integer and all features are single precision floats. Let $|X|$ denote the number of features in X. $|T_{str}|$ and $|T_{img}|$ are straightforward to calculate, since they are the base tables. For $|T_i|$ with feature layer $l = L[i]$, we have:

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

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

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

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

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

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

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

The other four variables can be set as derived variables. In the numerator, τ captures the relative total compute and communication costs, which are effectively a "constant" for this optimization. The second term captures disk spill costs for T_i's (with at least 50% of Core Memory being Storage Memory [6]). The denominator captures the degree of parallelism. While this objective is ideal, it is largely impractical and needlessly complicated for our purposes for three reasons. First, estimating τ is highly tedious, since it involves Spark shuffle costs, downstream model costs, etc. Second, and more importantly, we hit a point of diminishing returns with cpu_{spark} quickly, since CNN inference typically dominates total runtime and TF anyway uses all cores regardless of cpu_{spark}. That is, this workload's speedup against cpu_{spark} will be quite sub-linear (confirmed by Figure 10 in Section 5). Empirically, we find that about 7 cores typically suffice for Spark; interestingly, a similar observation is made in Spark guidelines [13, 15]. Thus, we cap cpu_{spark} at $cpu_{max} = 8$. Third, given this cap, we can just drop the term minimizing disk spill costs, since s_{double} will typically be smaller than the total memory, even after accounting for the CNNs due to the cap. Overall, these insights yield a much simpler objective that is still a reasonable surrogate

Table 2: Notation for Section 4 and Algorithm 1.

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

for minimizing runtimes:

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

The constraints for the optimization are as follows:

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

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

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

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

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

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

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

Equation 10 caps cpu_{spark} and leaves a CPU for the OS. Equation 11 captures User Memory needed for reading CNN models and invoking TF, copying materialized feature layers from TF, and holding M. Equation 12 is the Executor Heap Memory definition (Figure 3). Equation 13 constrains the total memory as per Figure 3; $cpu_{spark} \times |f|_{mem}$ is the Executor Native Memory used by TF. Equation 14 captures a Spark guideline for Core Memory [6]. Equation 15 requires n_p to be a multiple of the number of worker processes to avoid skews, while Equation 16 bounds the size of an intermedaite data partition, as per Spark guidelines [6].

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

Algorithm 1 The Vista Optimizer Algorithm.

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

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

5. EXPERIMENTAL EVALUATION

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

Datasets. We use two public real-world datasets: *Foods* [10] and *Amazon* [49]. *Foods* has about 20,000 examples with 130 numeric structured features such as nutrition facts along with pairwise/ternary feature interactions. An image of each food item is given in JPEG format. The target represents if the food is considered healthy. *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 (within top 100). We pre-processed the title string to extract 100 numeric features (an "embedding") using the popular Doc2Vec procedure [45]. We also convert the list of categories into 100 numeric features using PCA on the indicator vectors representing all categories. All images are resized to 227×227 resolution, as required by most popular CNN models. Our data pre-processing scripts will be released on our project webpage; we hope our efforts help spur more research on this topic.

Workloads. We use three popular ImageNet-trained deep CNNs: AlexNet [43], VGG16 [58], and ResNet50 [35], obtained from [4, 9]. They complement each other in terms of model size and total MFLOPs [28]. We select the following interesting layers for feature transfer from each (see Table 1 for layer sizes): *conv5* to *fc8* from AlexNet ($|L| = 4$); *fc6* to *fc8* from VGG ($|L| = 3$), and top 5 layers from ResNet (from its last two layer blocks [35]), with only the topmost layer being fully-connected. Following standard practices [17, 65], we apply max pooling on the convolutional feature layers to

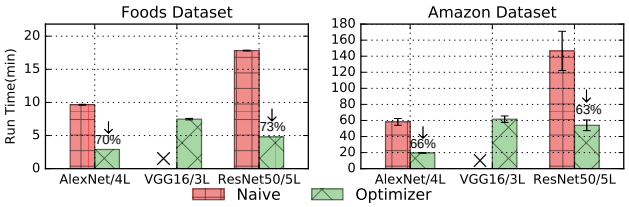


Figure 6: End-to-end reliability and efficiency. "×" indicates a system crash.

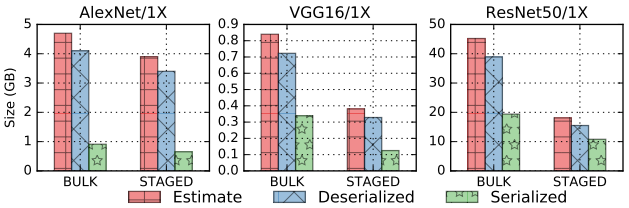


Figure 7: Size of largest intermediate table.

reduce their dimensionality before using them for M.[4] As for M, we run MLlib's logistic regression for 10 iterations.

Experimental Setup. We use a cluster with 8 worker nodes and 1 master in an OpenStack instance on CloudLab, a flexible and free compute resource for research [54]. Each node has 32 GB RAM, 8 VCPUs allocated from Intel Xeon @ 2.00GHz CPUs, 300 GB of secondary storage allocated from Seagate Constellation ST91000640NS HDDs. They run Ubuntu 16.04. We use Apache Spark v2.2.0 with *TensorFrames* v0.2.9 integrating it with TensorFlow v1.3.0. Spark runs in standalone mode and each worker runs one Executor. HDFS replication factor is three; input data is ingested to HDFS and read from there. Each runtime reported is the average of three runs with 90% confidence intervals shown.

5.1 End-to-End Reliability and Efficiency

We first validate if Vista improves overall reliability and efficiency for the above workloads. We compare two plans: *Naive* and Vista. *Naive* is the current dominant practice of running all feature transfer queries separately, as shown in Section 3.2, with Spark configured by standard practices [6, 13] (JVM heap size set to 29 GB, 5 CPUs used per Executor, *pers* set to memory-and-disk deserialized, *join* set to *shuffle*, and all other parameters set to default, including n_p and how the regions are split within the JVM heap). Vista is the plan picked by the Vista optimizer, including for system configuration (Section 4). Figure 6 shows the results.

We see that Vista improves both reliability and efficiency. With *VGG16*, *Naive* simply crashes on both datasets. This is due to blowups in Native Memory during the CNN inference, as explained in Section 4.1 (*VGG* has a memory footprint of 4 GB in the Spark-TF combine). However, Vista finishes in reasonable times on both datasets, which could reduce both user frustration and costs. As for the other cases where *Naive* does not crash, Vista improves efficiency significantly, reducing runtimes by 63%–73%. This reduction arises because Vista removes redundancy in partial CNN inference and chooses appropriate system parameters, as explained in Section 4.2. The reduction depends on the CNN and L: if more layers are explored, and if they are closer to the top, the more redundancy there is, and the faster Vista will be. Overall, these results confirm the benefits of an automatic optimizer such as Vista for improving reliability and efficiency of feature transfer workloads.

5.2 Drill-Down Analysis of Trade-offs

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

Intermediate Table Sizes. We first check if our optimizer's estimates of the intermediate table sizes are accurate. While it only uses Staged inference, we include Bulk inference for a comparison. Figure 7 shows the estimated and actual sizes. We see that the estimates are accurate for the deserialized in-memory data, with a reasonable safety margin. Interestingly, Bulk is not that much larger than Staged for AlexNet. This is because among its four layers explored, *conv5* is disproportionately large, while for the other two, the layer sizes are more comparable (see Table 1). Serialized is obviously smaller than deserialized, since Spark compresses the data. Interestingly, AlexNet feature layers seem more compressible; we verified that this was due to lots of zero values (caused by ReLU). On average, AlexNet feature layers had only 13.0% non-zero values, while VGG16's and ResNet50's had 36.1% and 35.7%, respectively. Exploiting such compression gains in our optimizer presents an interesting avenue for future work.

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

Physical Plan Decisions. We compare four combinations: shuffle ("SHUFFLE") or broadcast ("BROAD") join and serialized ("SER") or deserialized ("DESER") persistence format. We vary the data scale and the number of structured features ($|X_{str}|$) for both AlexNet and ResNet. The logical plan used is Staged/AJ. Figure 8 shows the results. We see that all four plans are almost indistinguishable for ResNet (Figure 9(B)), except at the 8X scale, the DESER plans outperform the SER plans. For AlexNet (Figure 9(A)) at the 8X scale, the SHUFFLE plans marginally outperform

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

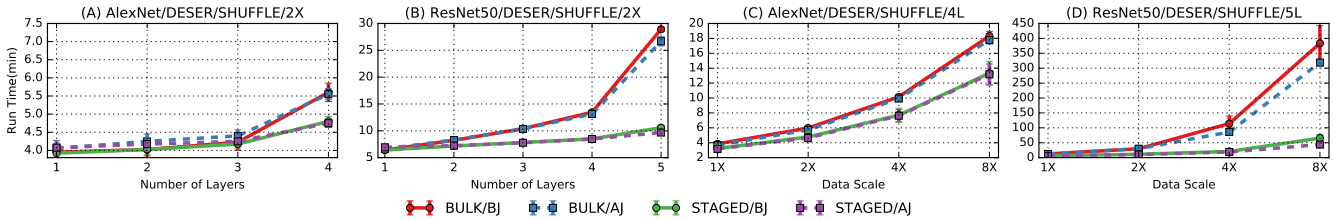


Figure 8: Runtime comparison of logical plan decisions for varying data scale and number of feature layers explored.

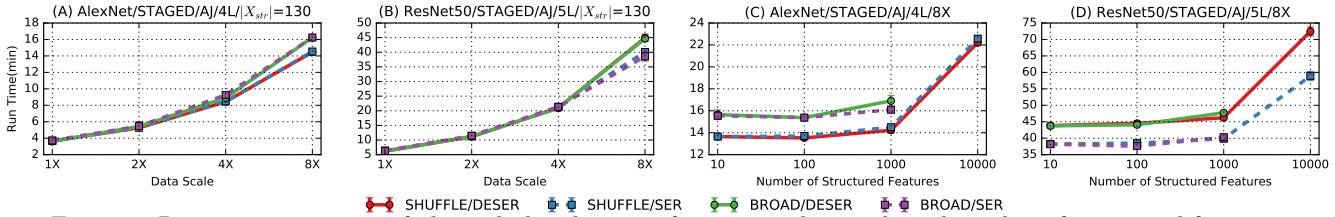


Figure 9: Runtime comparison of physical plan decisions for varying data scale and number of structured features.

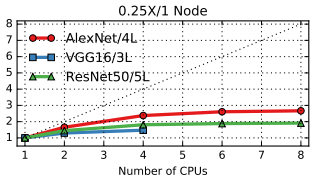


Figure 10: Speedup ratio for varying cpu_{spark} on one node.

the BROAD plans. Figure 9(C) shows this gap remains as $|X_{str}|$ increases, with the BROAD plans eventually crashing. For ResNet, however, Figure 9(D) shows that both SER plans are slightly faster than their DESER counterparts, but the BROAD plans still crash eventually. The gap between SER and DESER is significant for ResNet (but not AlexNet) because at the 8X scale, the largest intermediate table requires disk spills. Our optimizer accounts for such trade-offs automatically.

System Configuration Decisions. We now vary cpu_{spark} (with the optimizer setting the memory parameters accordingly) and n_p. The data scale is 1X; the plan is STAGED/AJ/SHUFFLE/DESER. Figures 11(A,B) show the results. As explained in Section 4.3, the runtime decreases with cpu_{spark} for all CNNs, but VGG eventually crashes (at 8 cores) due to the Native Memory blowup caused by the CNN replicas. The runtime decrease with cpu_{spark} is, however, sub-linear. To drill into this issue, we plot the speedup against cpu_{spark} on 1 node for data scale $0.25X$ (to avoid disk spills). Figure 10 shows the results: the speedups flatten out at about 4 cores. As mentioned in Section 4.3, this is to be expected, since CNN inference dominates total runtime and TF always uses all cores regardless of cpu_{spark}. Due to space constraints, we provide the runtime breakdowns in the technical report [20]. We also performed an experiment varying the JVM heap size; the runtimes did not change too much, unless the intermediate table is big enough to be spilled. Due to space constraints, we discuss this heap size experiment further in our technical report [20].

Figure 11(B) shows non-monotonic behaviors with n_p. If n_p is too small, the system crashes due to insufficient Core Memory for the join. It becomes feasible from $n_p = 8$. The runtimes go down, since we exploit more of the available parallelism (up to 32 cores usable by Spark). But eventually, the runtime rises again due to Spark overheads for handling too many tasks. In fact, when $n_p > 2000$, Spark compresses the task statuses sent to the master, which increases overhead substantially. By setting n_p automatically, Vista frees data scientists from having to navigate such trade-offs. In fact, our optimizer sets n_p at 160, 160, and 224 for AlexNet, VGG, and ResNet respectively, all of which yield runtimes comparable to the fastest runtimes seen here.

Scalability. Finally, we evaluate the speedup (strong scaling) and scaleup (weak scaling) of the plan STAGED/AJ/SHUFFLE/DESER as we vary the number of worker nodes (and also data scale for scaleup). While partial CNN inference and M are embarassingly parallel, data reads from HDFS and the join can bottleneck scaling behaviors. Figures 11 (C,D) show the results. We see near linear scaleup with all 3 CNNs. But Figure 11(D) shows that the AlexNet workload has a markedly sub-linear speedup, while VGG and ResNet exhibit near-linear speedups. To explain this difference, we drilled into the Spark logs and obtained the time breakdown for two major parts: 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. So, Spark overheads become non-trivial in AlexNet's case. Due to space constraints, we provide further analysis of the speedup results in the technical report [20].

5.3 Discussion and Limitations

Our experience with Vista reveals a pressing need to simplify the deployment of deep learning for data analytics. While TF is a poweful tool for training deep learning models, its poor support for data independence forces users to manually manage data files, distribution, memory, etc. This could be a hurdle for enterprise and domain scientific users. Spark has much better data independence, but poor support for deep learning. Thus, the Spark-TF combine is a powerful marriage of two complementary systems for unifying structured and unstructured data analytics. But as our work shows, this combine is still rudimentary and much work is still needed to improve system reliability, efficiency, and user productivity. Vista is a first step in this direction.

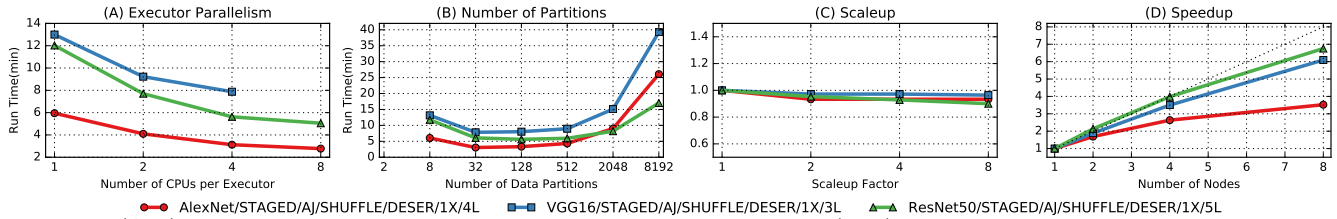


Figure 11: (A,B) Runtime effects of varying system configuration parameters. (C,D) Scaleup and speedup behavior on cluster.

Finally, we recap the assumptions and limitations of this work for the benefit of readers. Vista focuses on large-scale feature transfer from deep CNNs for multimodal data analytics combining images and structured data. It currently supports a roster of popular CNNs, not arbitrary CNNs. It also optimizes for the use of MLlib's logistic regression as the downstream ML model. Finally, we did not consider secondary storage space as a major concern. Nothing in Vista prevents its extension along any of these lines. For instance, one could add support for other downstream ML models by providing our optimizer estimates of its memory footprint. But it is more complicated to support arbitrary CNNs, since that requires static analysis of TF computational graphs. We leave such extensions to future work.

6. OTHER RELATED WORK

Multimodal Analytics. Transfer learning has been used for other forms of multimodal data analytics too, including image captioning [42]. Our focus is on the systems arising from integrating image and structured data features. A related but orthogonal line of work is "multimodal learning" in which a deep neural network (or another ML model) is trained from scratch on multimodal data [52, 59]. While feasible for some applications, this approach faces the same cost- and data-related 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 [24, 40]. They relied on older handcrafted features such as SIFT and HOG [30,48], not learned or hierarchical CNN features, although there is a resurgence of interest in CBIR with CNN features [63,66]. 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 recent video analytics system that uses deep CNNs to detect objects in video streams [41]. To reduce costs, NoScope builds a "cascade" of faster models, including smaller CNNs, that exploit temporal redundancy and information locality in video. Vista is orthogonal, since we focus on feature transfer; one could integrate our ideas with model cascades to improve efficiency further.

Query Optimization. Our work is inspired by a long line of work in the database query optimization literature, especially optimizing SQL queries with UDFs, multi-query optimization (MQO), and self-tuning DBMSs. For instance, [36] studied the problem of predicate migration to optimize the order of operations in complex relational query plans with multiple joins and UDF-based predicates, while [29] introduced faster algorithms and pruning techniques for finding the optimal plan for such queries with UDF predicates. Also related is [18], which studied "semantic" optimization of queries with predicates based on data mining classifiers. Unlike such works on optimizing queries with UDFs in the `WHERE` clause, our work can be viewed as optimizing CNN-based UDFs expressed as TensorFlow computational graphs within the `SELECT` clause for materializing CNN feature layers. We study and optimize novel materialization trade-offs for such queries. The new logical plans of Vista can be viewed as a form of MQO, which has been studied extensively for SQL queries [56]. By formalizing CNN inference operations, Vista is able to apply the general idea of MQO to complex CNN-based feature transfer queries and optimize its materialization trade-offs. Finally, there is much prior work on automatically tuning system parameters for relational and MapReduce workloads (e.g., [37, 62]). Our work is inspired by such systems, but we focus specifically on the emerging workload of feature transfer from deep CNNs and study the novel twists and systems trade-offs it presents.

7. CONCLUSIONS AND FUTURE WORK

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

As for future work, we plan to support newer and more general forms of CNNs and downstream ML tasks, as well as the interpretability of such models in data analytics contexts. We also plan to deepen the integration of deep learning models with data analytics systems to enable seamless and efficient multimodal data analytics that exploits other forms of unstructured data as well.

8. REFERENCES

[1] Apache spark: Lightning-fast cluster computing. `http://spark.apache.org`. Accessed December 31, 2017.

[2] Benchmarks for popular cnn models. `https://github.com/jcjohnson/cnn-benchmarks`. Accessed December 31, 2017.

[3] 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 December 31, 2017.

[4] Cafee model zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo. Accessed December 31, 2017.

[5] Deep learning with apache spark and tensorflow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html. Accessed December 31, 2017.

[6] 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 December 31, 2017.

[7] History of computer vision contests won by deep cnns. http://people.idsia.ch/~juergen/computer-vision-contests-won-by-gpu-cnns.html. Accessed December 31, 2017.

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

[9] Models and examples built with tensorflow. https://github.com/tensorflow/models. Accessed December 31, 2017.

[10] Open food facts dataset. https://world.openfoodfacts.org/. Accessed December 31, 2017.

[11] A peek at trends in machine learning by andrej karpathy. https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106. Accessed December 31, 2017.

[12] The small files problem of hdfs. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed December 31, 2017.

[13] Spark best practices. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/. Accessed December 31, 2017.

[14] Spark memory management. https://0x0fff.com/spark-memory-management/. Accessed December 31, 2017.

[15] Sparkdl: Deep learning pipelines for apache spark. https://github.com/databricks/spark-deep-learning. Accessed December 31, 2017.

[16] Tensorframes: Tensorflow wrapper for dataframes on apache spark. https://github.com/databricks/tensorframes. Accessed December 31, 2017.

[17] Transfer learning with cnns for visual recognition. http://cs231n.github.io/transfer-learning/. Accessed December 31, 2017.

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

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

[20] Materialization trade-offs for feature transfer from deep cnns for multimodal data analytics [technical report], Dec. 2017. https://adalabucsd.github.io/papers/TR_2017_Vista.pdf.

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

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

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

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

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

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

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

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

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

[30] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893. IEEE Computer Society, 2005.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. PRE MATERIALIZING A BASE LAYER

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

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

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

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

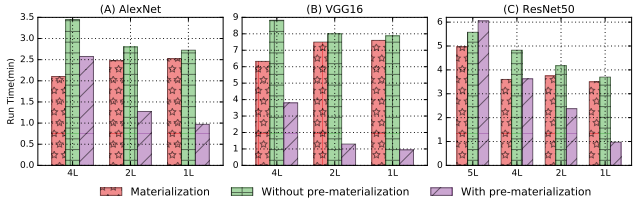
We perform a new set of experiments exploring different number of layers (say 4, 2, and 1 layers from the top separately) for each CNN model where the lowest layer is pre-materialized. Therefore for evaluating the downstream ML model for the lowest layer no CNN inference is required. For the other layers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

For AlexNet and VGG16 when exploring top 4, 2, and 1 layers the materialization time of the features of the lowest layer increases as evaluating higher layer requires more computations (see Figure. 12 (A) and (B)). However for ResNet50 there is a sudden drop in the materialization time of 5^{th} layer features to materialization time of 4^{th} layer features. This can be attributed to the disk IO overhead of writing out 5^{th} layer image features which is ∼3 times larger than that of layer 4 (see Figure. 12 (C)). Surprisingly we found that starting from a pre-materialized feature layer instead of raw images may or may not decrease the overall CNN feature transfer workload runtime. For AlexNet and VGG16, plan which starts from the pre-materialized base layer improves the runtime compared to the plan which

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

starts from raw images. The time reduction is high when materializing last layers (1^{st} and 2^{nd} layers from the top) where the size of the materialized features is small. But for ResNet50 this is not always the case. When performing CNN feature transfer for the top 5 layers, plan which starts from the pre-materialized layer performs worse than the plan which starts with raw images. This is because of the high IO overhead of reading large image features and sending them over the network (for the shuffle join) which is ∼44 times the size of raw images (see Figure. 3 (C)). In this case the computational time saved by materializing features from a base layer is dwarfed by the high IO overhead of handling large image features. Hence materializing CNN features from a base layer may not be advantageous always.

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

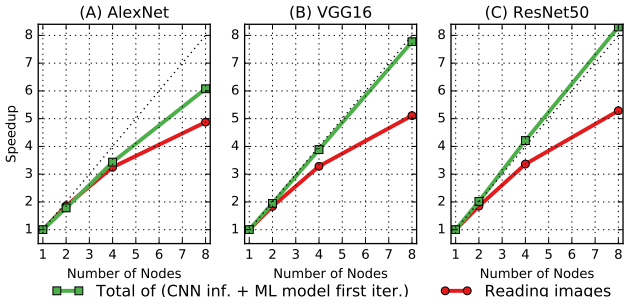


B. RUNTIME BREAKDOWN

In Figure. 11 (D) we see than all CNN models show sub-linear speedup behavior and speedup behavior of AlexNet is worse than other two models. However CNN inference is an embracingly parallel task and one would expect near linear speedups for CNN feature transfer workloads. To explain this peculiarity we drill-down into the time breakdowns of the workloads and explore where the bottlenecks occur. Typically Spark operations are run in pipelined fashion (except `mapPartition`, and `shuffling` operations). Therefore taking a precise time breakdown for all the sub-tasks is not possible. However Spark breaks down tasks into multiple stages when ever there is a shuffle boundary (e.g. a shuffle join) or into separate jobs when ever a Spark action (e.g `collect`, `count`) is being called. Therefore in the CNN feature transfer workloads that we explore reading of input data (structured data file and image files) and writing of shuffle-blocks will be separated into two sub stages. Also every iteration in the Logistic Regression (LR) model (downstream ML task) will also invoke a new job and the time consumed by each job can be obtained from the Spark Admin UI.

In the downstream Logistic Regression model, the time spent for training the model on features from a specific layer is dominated by the runtime of the first iteration. In the first iteration partial CNN inference has to be done starting either from raw images or from the image features from the layer below and the later iterations will be operating on top of the already materialized features. For the input reading the time is dominated by image data, because there are large number of small files compared to the structured file which is a one large file [12]. Table. 4 summarizes the time breakdown for the CNN feature transfer workloads when using different CNN models with the Foods Dataset. It can be seen that most of the time is spent on performing the CNN inference and LR 1^{st} iteration on the first layer (e.g 5^{th} layer from top for ResNet50) where the CNN inference has to be performed starting from raw images. Also it can

Figure 13: Drill-down analysis of Speedup Curves



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

We also analyze the speedup behavior for the input image reading and the sum of CNN inference and LR 1^{st} iteration times (see Figure. 13). Recall that for the overall speedup curves in Figure. 11 we observer sub-linear speedups for all three CNN models. However when we separate out the CNN inference plus LR 1^{st} iteration time we see slightly super linear speedups for ResNet50, near linear speedups for VGG16 and slightly better sub-linear speedups for AlexNet. Compared to AlexNet, VGG16 and ResNet50 CNN inferences are highly compute intensive (AlexNet, ResNet50 and VGG16 requires 725 MFLOPS, 7,754 MFLOPS and 18,721 MFLOPS respectively) and therefore shows better speedup behavior. For ResNet50 the intermediate data sizes are large and when we increase the number of nodes the amount of disk spills decreases. Hence we see slightly super linear speedups for ResNet. For reading input images we see sub-linear speedups which is bottlenecked by HDFS.

C. HEAP SIZE & CPUS PER EXECUTOR

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

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

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

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