

Snap Machine Learning

Celestine Dünner*, Thomas Parnell*, Dimitrios Sarigiannis, Nikolas Ioannou, Haralampos Pozidis

IBM Research

Zürich, Switzerland

{cdu,tpa,rig,nio,hap}@zurich.ibm.com

ABSTRACT

We describe an efficient, scalable machine learning library that enables very fast training of generalized linear models. We demonstrate that our library can remove the training time as a bottleneck for machine learning workloads, opening the door to a range of new applications. For instance, it allows more agile development, faster and more fine-grained exploration of the hyper-parameter space, enables scaling to massive datasets and makes frequent re-training of models possible in order to adapt to events as they occur. Our library, named Snap Machine Learning (Snap ML), combines recent advances in machine learning systems and algorithms in a nested manner to reflect the hierarchical architecture of modern distributed systems. This allows us to effectively leverage available network, memory and heterogeneous compute resources. On a terabyte-scale publicly available dataset for click-through-rate prediction in computational advertising, we demonstrate the training of a logistic regression classifier in 1.53 minutes, a 46x improvement over the fastest reported performance.

KEYWORDS

Distributed System, Heterogeneous Systems, Machine Learning, GPU acceleration

1 INTRODUCTION

The widespread adoption of machine learning and artificial intelligence has been, in part, driven by the ever-increasing availability of data. Large datasets enable training of more expressive models, thus leading to higher quality insights. However, when the size of such datasets grows to billions of training examples and/or features, the training of even relatively simple models becomes prohibitively time consuming. This long turn-around time (from data preparation to inference) can be a severe hindrance to the research, development and deployment of large-scale machine learning models in critical applications.

However, it is not only large data applications in which training time can become a bottleneck. For example, real-time or close-to-real-time applications, in which models must react rapidly to changing events, are another important scenario where fast training times are vital. For instance, consider security applications in critical infrastructure when a new, previously unseen, phenomenon is currently evolving. In such situations, it may be beneficial to train, or incrementally re-train, the existing models with new data. One's ability to respond to such events necessarily depends on the training time, which can become critical even when the data itself is relatively small.

A third area when fast training is highly desirable is the field of ensemble learning. It is well known that most data science

competitions today are won by large ensembles of models [16]. In order to design a winning ensemble, a data scientist typically spends a significant amount of time trying out different combinations of models and tuning the large number of hyper-parameters that arise. In such a scenario, the ability to train models orders of magnitude faster naturally results in a more agile development process. A library that provides such acceleration can give its user a valuable edge in the field of competitive data science or any applications where best-in-class accuracy is desired. One such application is click-through rate prediction in online advertising, where it has been estimated that even 0.1% better accuracy can lead to increased earning of the order of hundreds of millions of dollars [9].

A growing number of small and medium enterprises rely on machine learning as part of their everyday business. Such companies often lack the on-premises infrastructure required to perform the compute-intensive workloads that are characteristic of the field. As a result, they may turn to cloud providers in order to gain access to such resources. Since cloud resources are typically billed by the hour, the time required to train machine learning models is directly related to outgoing costs. For such an enterprise cloud user, the ability to train faster can have an immediate effect on their profit margin.

The above examples illustrate the demand for fast, scalable, and resource-savvy machine learning libraries. Today there is an abundance of general-purpose environments, offering a broad class of functions for machine learning model training, inference, and data manipulation. Some of the most prominent and broadly-used ones are listed below, along with certain advantages and limitations.

- *scikit-learn* [12] is an open-source module for machine learning in python. It is widely used due to its user-friendly interface, comprehensive documentation the wide range of functionality that it offers. While scikit-learn does not natively provide any GPU support, it can call lower-level native C++ libraries such as LIBLINEAR to achieve high-performance. A key limitation of scikit-learn is that it does not scale to datasets that do not fit into the memory of a single machine.
- *Apache MLlib* [10] is Apache Spark's** scalable machine learning library. It provides distributed training of a variety of machine learning models and provides easy-to-use APIs in Java**, Scala and Python. It does not natively support GPU acceleration, and while it can leverage underlying native library like BLAS for certain operations, it tends to exhibit slower performance relative to the same distributed algorithms implemented natively in C++ using high performance computing frameworks such as MPI [2].
- *TensorFlow* [1] is an open source software library for numerical computation using data flow graphs. While TensorFlow** can

*Equal contribution.

be used to implement algorithms at a lower-level as a series of mathematical operations, it also provides a number of high-level APIs that can be train generalized linear models without needing to implement them oneself. It transparently supports GPU acceleration, multi-threading and can scale across multiple nodes. When it comes to training of large-scale linear models, a downside of TensorFlow is the relatively limited support for sparse data structures, which are frequently important in such applications.

In this work we describe a new library that exploits the hierarchical memory and compute structure of modern systems. We focus on the training of generalized linear models. We combine recent advances from algorithm and system design to optimally leverage all hardware resources available in modern computing environments. The three main features that distinguish our system are *distributed training*, *GPU acceleration* and full support of *sparse data structures*:

- *Distributed Training* We build our system as a data-parallel framework. This enables us to scale out and train on massive datasets that exceed the memory capacity of a single machine which is crucial for large-scale applications. To guarantee good performance in a distributed setting we build on a state-of-the-art framework that achieves communication-efficient training, respects data-locality and provides adaptivity to diverse system characteristics.
- *GPU Acceleration*. To speed up training algorithms we support GPU accelerators in a distributed environment. Such specialized hardware devices are widely available in today’s cloud offerings and have attracted a lot of attention especially in the context of training deep neural networks. We build on specialized solvers designed to leverage the massively parallel architecture of GPUs. By offloading the entire solver to the GPUs we avoid large data transfer overheads and respect data locality in GPU memory. To make this approach scalable we take advantage of recent developments in heterogeneous learning in order to enable GPU acceleration even if only a small fraction of the data can indeed be stored in the accelerator memory.
- *Sparse data structures* Many machine learning datasets are sparse. Hence the support of sparse data structures is essential pillar of our library. In this context, we detail some new optimizations for the algorithms used in our system when applied to sparse data structures.

2 SYSTEM DESCRIPTION

We start with a high-level, conceptual description of the structure of Snap ML and detail individual components in the subsequent sections 3 and 4. Our system implements several hierarchical levels of parallelism in order to partition the workload among different nodes in a cluster, take full advantage of accelerator units and exploit multi-core parallelism on the individual compute units.

2.1 1st Level Parallelism

The first level of parallelism spans across individual worker nodes in a cluster. The data is stored distributedly across multiple worker nodes that are connected over a network interface as shown in Figure 1. This data-parallel approach enables the training on large-scale datasets that exceed the memory capacity of a single device.

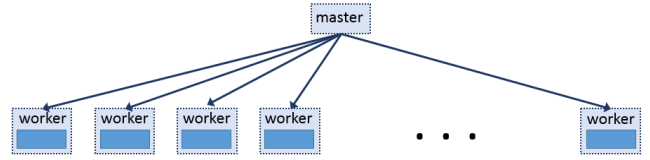


Figure 1: Data-parallelism across worker nodes in a cluster

2.2 2nd Level Parallelism

On the individual worker nodes we can leverage one or multiple accelerator unit such as, e.g., GPUs, by systematically splitting the workload between the host and the accelerator units. The different workloads are then executed in parallel enabling full utilization of all hardware resources on each worker and hence achieving the second level of parallelism across heterogeneous compute units illustrated in Figure 2.

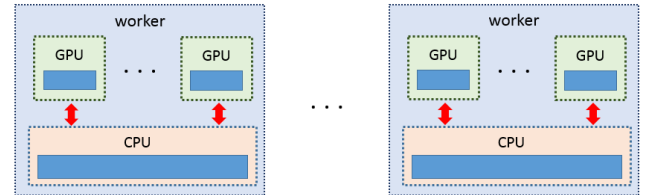


Figure 2: Parallelism across heterogeneous compute units within one worker node.

2.3 3rd Level Parallelism

In order to efficiently execute the workloads assigned to the individual compute units we leverage the parallelism offered by the respective compute architecture. We use specially designed solvers to take full advantage of the massively parallel architecture of modern GPUs and implement multi-threaded code for processing the workload on the CPU. This results in the additional, third level of parallelism across cores as illustrated in Figure 3.

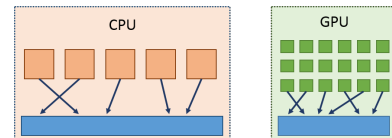


Figure 3: Multi-Core Parallelism within individual compute units

3 ALGORITHMIC CORE

In the following we describe the algorithmic backend of our system in more detail. The core innovation relates to how we nest multiple state-of-the-art algorithmic building blocks to reflect the hierarchical structure of a distributed architecture.

3.1 Distributed Framework

The first level of parallelism serves mainly to increase the overall memory capacity of our system in order to enable scaling to massive datasets. In such a data-parallel setting the data is partitioned across K worker nodes. Hence, every worker node has only access to its local partition of the training data and the main challenge of designing an efficient distributed algorithm relates to the high cost of exchanging information over the network between the nodes. Of particular interest are methods that allow flexibility to adjust to various systems characteristics such as communication latency and computation efficiency as studied in [2]. This is typically achieved by introducing a tunable hyper-parameter to steer the amount of work that is performed between two consecutive rounds of communication.

CoCoA. One distributed algorithmic framework that offers such a adjustable hyper-parameter is CoCoA [13]. We refer to it as a framework rather than an algorithm because it defines how to split the workload between the different nodes and which information to exchange between nodes in order to synchronize their work, but it does not dictate which solver should be used on the individual worker nodes to solve their local optimization problem. This allows us to take advantage of the available compute and memory resources when choosing a solver. Hence, CoCoA is perfectly suited to reflect the first level of parallelism of our system where the implementation of the local solver is handed over to a higher hierarchical level of the system. A known limitation of the CoCoA framework is its bad performance for non-quadratic loss functions such as logistic regression; this can be attributed to the quadratic approximation of the loss function used to construct the local subproblems. To improve the performance for these particular applications we modify the local subproblems in CoCoA to incorporate local second-order information. For implementation details of CoCoA we refer to Section 4.

3.2 Heterogeneous Nodes

The second level of parallelism is implemented to take advantage of one, or potentially multiple, accelerator units available on the individual worker nodes to solve the local optimization task in the CoCoA framework. How we integrate these accelerator units in the optimization procedure depends on the locally available accelerator memory. Therefore we distinguish three application scenarios as illustrated in Figure 4a, i.e.,

- (1) aggregated GPU memory \geq local training data
- (2) aggregated GPU memory $<$ local training data
- (3) no GPU available

Scenario (1). If the aggregated GPU memory can fit the entire dataset we partition the local training data evenly across the available GPU accelerators. Then, the data resides locally in the accelerator memory and the individual GPUs are treated as independent workers and a second level of CoCoA implemented to solve the local subproblems in a distributed fashion. This nested structure reflects the hierarchical structure of the distributed system and we can account for the fact that the communication within a node is less expensive than communication across the network by enabling multiple communication rounds on the inner level within a single communication round across the network.

Scenario (2). This is the most common setting encountered in distributed systems but also the most challenging one because the local training data can not be stored entirely in GPU memory. To nevertheless take full advantage of the GPU accelerators we build the local solver on the recently developed DuHL scheme [3]. DuHL provides a systematic way to split the workload between the host CPU unit, storing the local data in memory, and the accelerator units being attached to it by respecting the tight memory constraint of the accelerator. This results in parallelizable tasks that can be executed simultaneously. More details about the implementation and the extension to sparse data structures are given in Section 4.

Scenario (3). In the case where each worker has only a single CPU, there is no parallelism across compute units to exploit at this level. However, for worker machines that have a multi-socket architecture with multiple CPUs, there is typically a cost associated with inter-socket communication. Furthermore, the issue of false sharing can arise when threads on different sockets attempt to write to the same location in memory. These effects mean that the multi-threaded solvers detailed in the next section cannot be readily applied across sockets. To solve this problem, one can implement CoCoA across the sockets. In this manner, there is a separate local solver running on the cores of each socket, and the communication between them can be effectively controlled in a synchronous manner.

We note that for scenario (1), (2) and (3) we have orchestrated the work between the individual compute nodes within a single worker but we have not specified the algorithm that is implemented on the GPU (and the CPU) to solve its local optimization task. This is implemented by the third hierarchical level of our system.

3.3 Multi-Core Parallelism

The third level of parallelism concerns the parallelism across different cores of the CPU and/or the GPU.

GPU solver. To efficiently solve the optimization problem assigned to the GPU accelerator we implement the twice parallel asynchronous stochastic coordinate descent solver (TPA-SCD) [11]. It maps a stochastic coordinate solver to the massively parallel architecture of modern GPUs by leveraging the different levels of parallelism, the shared memory available within multi-processors and the built-in atomic add operation. This solver can be used as a stand-alone solver, as a local solver within the CoCoA framework as in scenario (1) or as part of DuHL in a heterogeneous setting

as in scenario (2). In Section 4 we discuss the implementation of this solver in more detail.

CPU solver. The CPU can take two roles, depending on which of the three scenarios, discussed in the previous section, applies.

- (1) If the entire local partition of the training data can be distributed among the GPUs, we ship all the work to the GPUs [11]. In this case the CPU takes the role of the master node in the second level of CoCoA and manages the communication and synchronization of the work between the GPU accelerators.
- (2) If the CPU is used as part of DuHL it is assigned the task of identifying the subset of the training data to be processed on the GPU for the next round. This consists of randomly sampling datapoints and computing the respective importance value to the optimization task. This computation is independent for individual datapoints and can thus be parallelized over the available cores of the CPU by assigning a subset of the datapoints to each core. This allows full utilization of the compute power of the CPU in parallel to the GPU as discussed in the previous section.
- (3) If no GPU is available the CPU will run the local optimization task. Therefore we use PASSCoDe [6], a parallel solver designed for multi-core CPUs. For implementation details see Section 4.5.

4 SYSTEM IMPLEMENTATION

The implementation of our system reflects the nested hierarchical structure of the algorithmic core illustrated in Figure 6. At the heart of our system is a C++/CUDA library that implements the computational heavy and performance critical parts of the system in level 2 and level 3. The data-parallelism on the outer most level can be implemented on top of any distributed computing framework where the individual workers call the underlying library - we will discuss Spark and MPI in more detail. To detail the implementation challenges we will again proceed level-by-level:

4.1 Level 1 Parallelism

The distributed framework implementing the first level of parallelism handles the data-partitioning and the communication of updates over the network between different nodes in the CoCoA framework. In order to satisfy needs of different users we have implemented our system on top of two different distributed frameworks - one being the widely-used Apache Spark framework [20] and the other being the high-performance computing framework MPI [5]. In the following we will discuss these two implementations separately.

Apache Spark. To seamlessly integrate our system in Spark-based applications we provide an implementation of our system on top of Apache Spark. This concerns the outer most level of CoCoA. To maintain good performance on top of Spark we have adopted the implementations and optimizations for CoCoA proposed in [2], i.e., we implemented

- *meta-RDDs* to avoid large overheads of Spark related to data management and enable the use of GPUs within the Spark framework.
- *persistent local memory* to avoid unnecessary communication overheads and keep the local model parameters in memory on the worker nodes.

As noted in [2] these optimizations are not fully compatible with the Spark programming model and break the data resiliency. However, for our performance optimized system, training times are very short; even for large-scale data training is in the order of minutes (see Section 5). Thus the overheads introduced by restarting the training procedure if a node fails are negligible. It would also be relatively straightforward to store a snapshot of model on disk in regular intervals and then reload this model together with the required data partitions and use it as warm-start for a new run. To provide built in data resiliency is left for future work.

MPI. We also built an implementation of our system entirely in C++ on top of the high performance MPI framework. To initially distributed the training data we use a load-balancing scheme to distribute the computational load evenly across worker nodes as in [2]. Regarding the communication of the update vectors in CoCoA, MPI offers more flexibility in designing the communication pattern, i.e., instead of centralized communication we can also use MPIs *AllReduce* primitives and achieve all-to-all communication among the nodes.

4.2 Level 2 Parallelism – Scenario (1)

In scenario 1 where the aggregated accelerator memory on a single node is large enough to store the local data partition we use the CoCoA framework to orchestrate work between the different accelerator units. This second level of CoCoA is implemented inside the C++ core of our library using asynchronous functionality provided by the CUDA framework for dealing with multiple GPUs. This is in contrast to how CoCoA is implemented in Level 1, where Spark or MPI was used, since it is a common scenario that one may want to leverage multiple GPUs in one machine without incurring the complexity and overheads of distributed computing frameworks.

4.3 Level 2 Parallelism – Scenario (2)

For scenario (2) we implement DuHL to distributed the workload between the CPU and the attached GPUs. For a detailed description of the algorithm procedure we refer to [3]. In the following we will detail the implementation of DuHL, propose some novel extensions to address sparse data structures and discuss several performance optimizations. The core of the DuHL scheme is a logic to repeatedly determine the subset of the training data for the next round, manage the content of the GPU memory and coordinate the work between host and accelerator units to avoid idle times.

Multiple GPUs. The DuHL scheme in [3] is proposed for a single accelerator unit, however, in modern servers we can often find multiple GPU accelerators per node. Thus, we have extended the DuHL scheme to take advantage of multiple accelerator units by combining it with CoCoA. We use CoCoA to distribute the local workload among the available accelerator units and then assign the

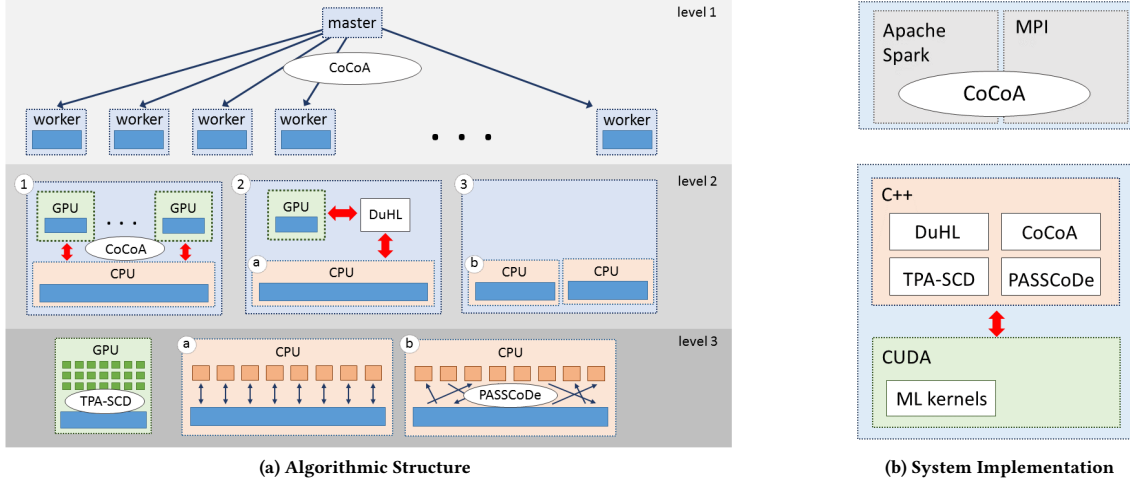


Figure 4: Hierarchical structure of our distributed system.

CPU cores evenly among them to run the DuHL scheme for every GPU accelerator on the dedicated local subproblem.

GPU Memory Management. For dense data structures, data vectors are of fixed size and the management of the GPU memory is fairly easy – the CPU only needs to keep track of the indices of the data vectors currently residing in the GPU memory. Then, every time an optimization round on the GPU has finishes, it sorts the importance values, finds to new index set, compares it to the old index set and swaps (i.e., over-writes) data vectors if necessary. However, for sparse data structures this is more challenging since we need to deal with the variable size memory allocation problem. Therefore use a custom memory pool allocator [8] that partitions the memory into fixed-sized blocks and uses offset-based linked-lists to keep track of the data vector location and status (free, used). Then, to update the GPU memory in every round, we iteratively delete less important data vectors and fill the available memory space with more important data vectors until the data vectors on the GPU represent the set of most important ones.

Data-Batching. To avoid large data transfer and memory management overheads when updating the GPU memory after every round, we group multiple training data vectors into batches and treat them as a single data unit. The importance of such a batch is then determined by the aggregated importance value of the vectors composing the batch. This modification theoretically preserves the superior convergence properties of the DuHL scheme over its random counterpart where batches are sampled uniformly at random. We found this modification to be necessary for sparse data structures in order to achieve good performance.

Workload Synchronization. To avoid idle times on the GPU we stop the importance value computation on the CPU as soon as the GPU has finished. We achieve this using CUDA events as proposed by the authors in [3].

Sequential Alternative. DuHL works particularly well for sparse models such as SVM or Lasso, however for dense models such as

logistic regression, we were, for certain datasets not always able to see a significant convergence gain of using the DuHL. While we need to further investigate the origin of this performance drop, and adapt DuHL accordingly, we provide an optimized sequential alternative to DuHL. That is, we process data sequentially in batches – this can be viewed as DuHL with a degenerate importance function returning the time passed since the data vector has been processed the last time. This sequential approach does not enjoy the favorable convergence behavior of DuHL, however we still fully utilize all resources by assigning an alternative workload to the CPU – the CPU is used to perform permutations of future batches to be processed on the GPU. This split of workload has the additional advantage that data transfer overheads can fully be interleaved with the computational workload. To achieve this we make use of CUDA streams where one stream is assigned to copying data on the GPU for the next iteration while the other stream executes the computational workload. We will analyze the performance of this approach in more detail in Section 5.4.

4.4 Level 2 Parallelism – Scenario (3)

Currently, inter-socket CoCoA is not explicitly implemented in Snap ML. We defer the problem to the top-level of parallelism and just spawn a separate MPI process or Spark executor that is pinned to each socket. While this approach is acceptable, we acknowledge that since inter-socket communication is typically much cheaper than communication over the network, there may be some potential benefit to communicating more frequently between the workers on each socket relative to the workers on different machines. Therefore, we plan to implement inter-socket CoCoA explicitly using multi-threaded C++ code in the near future.

4.5 Level 3 Parallelism

GPU solver. To benefit of the parallelism offered by GPUs when solving the local optimization problem we implement the TPA-SCD

solver as detailed in [11] and [3]. TPA-SCD defines a stochastic coordinate descent, in which every coordinate update is assigned to a different thread block for asynchronous execution on the streaming multi-processors of the GPU. Within each thread block, the inner products that are essential to solve the coordinate-wise subproblems are computing using warps of tightly-coupled threads. In the previous literature, TPA-SCD was applied to objective functions such as ridge regression[11] and support vector machines [3], both of which have the desirable property that the objective function can be minimized exactly with respect to a single model coordinate while keeping the other coordinates fixed. In Snap ML, we also support objective functions for which this is not the case such as the dual form of logistic regression. To address this issue, instead of solving the coordinate-wise subproblem exactly, we make a single step of Newton’s method, using the previous value of the model as the initial point. We find that the computations required to compute the Newton step (i.e., the first and second derivative) and also be expressed in terms of a simple inner product and thus all of the same TPA-SCD machinery can be applied. In the broader context of our system, these GPU solvers can be used as a stand-alone solver, as a local solver in a distributed framework as in scenario (1), or as part of DuHL as in scenario (2).

CPU solver. For scenario (3) where the optimization problem is solved on the CPU we use the PASSCoDe algorithm [6]. The authors propose three different version of PASSCoDe handling the synchronization differently, these implementations were studied and compared in [11]. Motivated by these results we have implemented the PASSCoDe-atomic version of the PASSCoDe algorithm as a default solver. It is not as fast as the Passcode-wild alternative but its convergence behavior is more well defined.

5 EXPERIMENTAL RESULTS

In the following we will analyze the performance of Snap ML in different hardware scenarios and benchmark its performance against sklearn, TensorFlow and Apache Spark.

Application. For the experimental section we focus on the machine learning application of click-through rate prediction (CTR), which is one of the central uses of machine learning in the internet. CTR is a massive-scale binary classification task where the goal is to predict whether or not a user will click on an advert based on a set of anonymized features. For our experiments we use the publicly available Terabyte Click Logs dataset recently released by Criteo Labs [7]. It consists of a portion of Criteo’s traffic over a period of 24 days where every day an average of 160 million examples were collected - an example corresponds to a displayed ad served by Criteo and is labeled by whether or not this advert has been clicked. We use the data collected during the first 23 days for the training of our models and the last day for testing. For single node experiments in Section 5.1 as well as the analysis of individual components of Snap ML in Section 5.2 we will use smaller versions of this dataset such as i) the one released by Criteo Labs as part of their 2014 Kaggle competition where we perform a random 75%/25% train/test split and ii) a custom subsampled version of the Terabytes Click Logs data where we use the first 1 billion examples for training and the next 100 million examples for testing. For the different datasets are

specified in Table 1. Note that different pre-processing was used for the different versions of the dataset.

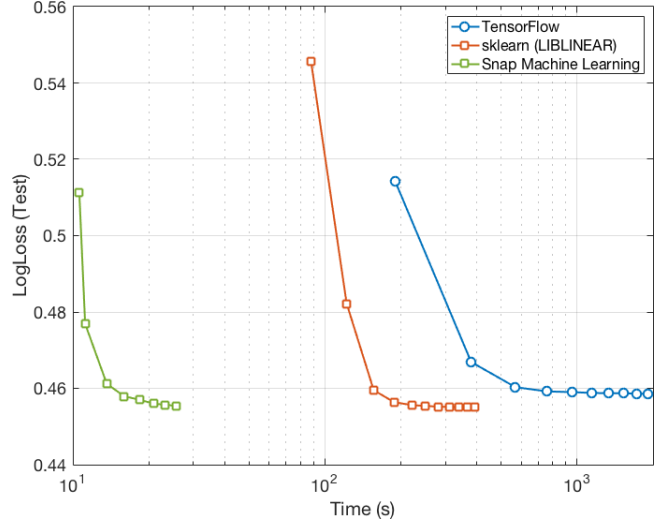


Figure 5: Single-node benchmark for criteo-kaggle dataset.

5.1 Single-Node Performance

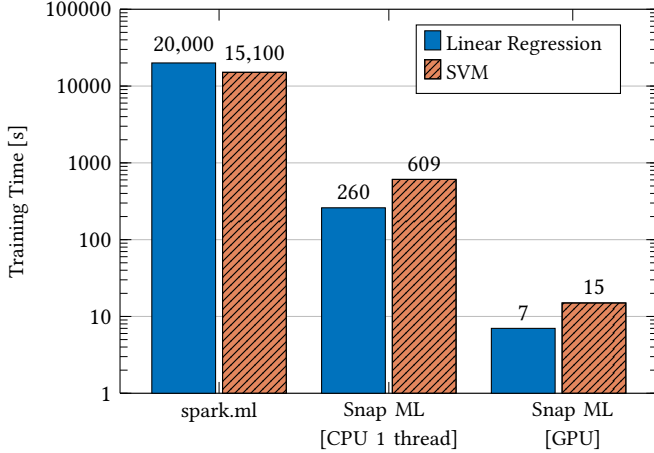
We start with an evaluation of the single node performance of Snap ML. We use the criteo-kaggle dataset. It is 11GB in size and can thus fit inside the memory of a single machine and even a single modern GPU. For this experiment we use an IBM Power System* AC922 server with four NVIDIA Tesla V100 GPUs attached via the NVLINK 2.0 interface. We only use one of the GPUs in this analysis. We benchmark the single node performance of Snap ML for the training of a Logistic Regression classifier against an equivalent solution in sklearn and TensorFlow. The same value of the regularization parameter ($\lambda = 10$) was used in all cases. The different frameworks are used as follows:

sklearn.linear.model.LogisticRegression. We load the data using the svmlight reader provided by sklearn, and then pickle the resulting data structures so that the data is stored on disk in a binary format. This allows us to read the data in few seconds each time we run the training rather than parsing the text-based svmlight files every time. To train the model we use the standard LogisticRegression class from sklearn, with the option to solve the dual formulation enabled which allows faster training for this application. Under the hood, sklearn is calling the LIBLINEAR library [4] to solve the resulting optimization problem. This library is written in C++ and designed to provide high-performance training of linear models with millions of training examples and features. It operates in single-threaded mode and does not leverage any available GPU resources.

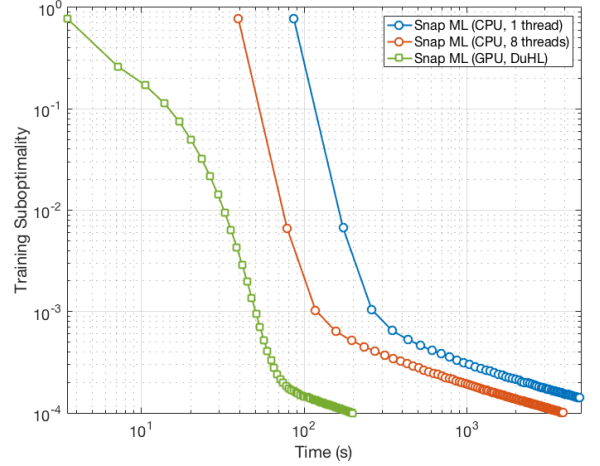
tf.contrib.learn.LinearClassifier. For the TensorFlow (TF) experiment, we first convert the svmlight data into the native binary format for TensorFlow (TFRecord) using a custom parser. The parser converts each example into appropriate key to TF feature

| | #features | #examples (train) | #examples (test) | size train [binary] | size train [libsvm] | pre-processing |
|----------------------------|------------|----------------------|---------------------|------------------------|------------------------|--|
| <i>criteo-kaggle</i> | 1 million | 34.4 million | 11.5 million | 11 GB | 19 GB | data as provided on the LIBSVM webpage [18] |
| <i>criteo-1b</i> | 10 million | 1 billion | 100 million | 98 GB | 274 GB | custom one-hot encoding of categorical variables |
| <i>Terabyte Click Logs</i> | 1 million | 4.2 billion | 180 million | 641 GB | 2.3 TB | data as provided on the LIBSVM webpage [18] |

Table 1: Specification of CTR datasets used for the experiments.



(a) Training Time for tera-scale GLMs in Spark MLlib vs. Snap ML.



(b) Training Time of tera-scale SVM in Snap ML on a Limited GPU Memory cluster.

Figure 6: Multi-Node Performance of training in Snap ML

pairs: one pair containing the label and one with the sparse feature indices. We then feed the TFRecord to a TF binary classifier (`tf.contrib.learn.LinearClassifier`), treating the TFRecord features as sparse columns with integerized features. We use the stochastic dual coordinated ascent optimizer provided by TF (`tf.contrib.learn.optimizer.SDCAOptimizer`), using the optimizer and train input function options suggested by Google [17]. The Experiment (`tf.contrib.learn.Experiment`) TF class is then used to train and evaluate the dataset, with custom input functions for training and evaluation. These input functions are responsible for reading the dataset, and transforming it to a tuple of features, label tensors that are then supplied to the classifier. Our input functions support batching and are multi-threaded, built on top of an existing TF API (`tf.contrib.learn.io.read_keyed_batch_features`). We use a batch size of 1M, and a number of IO threads equal to the number of physical processors – settings which we have experimentally found to perform the best. The implementation is multi-threaded and can leverage GPU resources (for the classifier training and evaluation) if available. In this case we let TensorFlow use a single V100 GPU since we found it was faster than using all four.

Snap.ml.LogisticRegression. We call Snap ML directly from Python, passing pointers to the data stored in scipy sparse data structures into the underlying C++ library. From there, the data is copied into the GPU memory in full, and the training is performed using TPA-SCD GPU solver on the dual problem.

In Figure 5, we compare the performance of the three aforementioned solutions. TensorFlow converges in approximately 500 seconds whereas sklearn takes around 200 seconds. This difference may be explained by the fact that sklearn keeps the entire dataset in memory whereas TensorFlow processes data in batches reading from disk each time. Finally we can see that Snap ML converges in around 20 seconds, an order of magnitude faster than the other frameworks. This speed-up can be attributed to the use of the novel GPU solver.

5.2 Multi-Node Performance

In order to evaluate and analyze the performance of Snap ML for CTR when scaled out across multiple machines, we use the *criteo-1b* dataset in this section which is 174 GB in size and thus cannot be loaded in the memory of a single machine. We conduct experiments for both application scenarios detailed in Section 3.2; (1) training on a cluster where the available GPU memory on every node fits the local training data and (2) training on a cluster where the available GPU memory can not fit the local training data. We will start with a comparison of Snap ML against Spark MLlib for a setting fitting scenario (1) and then analyze Snap ML’s performance in scenario (2).

5.2.1 Comparison with Spark MLlib. MLlib is Apache Spark’s scalable machine learning library. We compare the performance of Snap ML against Spark MLlib for training both a Linear Regression

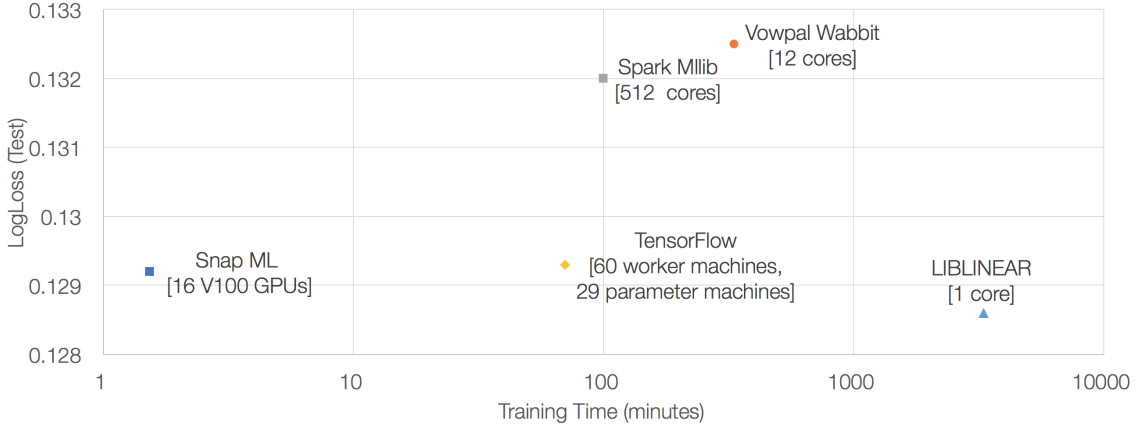


Figure 7: Comparison with previously-published results for Logistic Regression on the Terabyte Click Logs dataset.

(LR) model as well as a Support Vector Machine (SVM) model on the criteo-1b dataset. We deploy Spark on two IBM Power System S822LC servers with 8 executors (4 on each node). We load the training data using the libsvm reader provided by Spark and then train both the LR model (provided by spark.ml) and the SVM model (provided by spark.ml). We measuring the training time as well as the mean squared error on the test set (for regression) and the F1-score on the test set for the SVM.

We deploy Snap ML using the Apache Spark API that has been developed. Spark is thus used to implement the top-level parallelism (i.e., the CoCoA framework), while the local sub-problem on each executor is off-loaded to the core engine (written in C++), which can take advantage of lower-level parallelism if GPUs are available. Snap ML was deployed on the same servers that were used to evaluate Spark MLib. These servers have 4 NVIDIA Tesla P100 GPUs present which have 16GB of memory each, thus, fitting the entire training data. We measure the training time and MSE/F1-score for both the LR and the SVM that are provided by Snap ML, with and without GPU acceleration enabled (for this comparison we opt to use the single threaded CPU solver in Snap ML).

In Figure 6a, we present the training time of the aforementioned experiments. Without enabling the GPU acceleration, we observe more than an order of magnitude speed-up vs. MLib for both models: 77x for LR and 24x for SVM. When enabling the GPU acceleration, we obtain 3 orders of magnitude performance improvement relatively to MLib: 2800x for LR and 1000x for SVM. The MSE for the LR was the same in all experiments (6.9%), while for the SVM Spark obtained a slightly worse F1-score (0.13) relative to Snap ML (0.17). Since both models were configured with the same regularization parameter, this difference may be explained by some additional feature scaling that MLib is doing internally.

5.2.2 Limited-Memory Training. To analyze the performance of Snap ML in scenario (2) we train an SVM classifier on the criteo-1b dataset. We use a cluster of 4 nodes where each node has 2 NVIDIA GTX 1080 Ti GPUs attached. These GPUs have 11GB of memory of which 8GB can be used to store the data since we also need to store the model and auxiliary data structures. Hence the entire training data which is 98GB in size cannot be stored entirely inside the

GPU memory. To take advantage of both GPUs on every node for training the SVM model Snap ML uses the combination of DuHL with CoCoA described in Section 4.3. We compare the training time of the GPU accelerated version with the single-threaded, as well as the multi-threaded CPU solvers implemented in Snap ML. Results are reported in Figure 6b. We can see that even when the dataset does not fit in GPU memory, the use of DuHL allows us to benefit from GPU acceleration resulting in an order of magnitude faster training relative to both single-threaded and multi-threaded CPU solvers.

5.3 Tera-Scale Benchmark

For the large scale benchmark on the full dataset of 4.2 billion training examples, we deploy Snap ML across four IBM Power System AC922 servers. Each server has 4 NVIDIA Tesla V100 GPUs which communicate the the host via the NVLINK 2.0 interface. We use the MPI extensions to Snap ML so that the model is trained using 16 MPI processes, each process is assigned a unique GPU and is pinned to the corresponding NUMA socket where the GPU is attached. When training a Logistic Regression (LogR) model, we obtain a logarithmic loss on the test set of 0.1292 in 1.53 minutes. This is the total runtime including data loading, initialization and training time.

There have been a number of previously-published results on this same benchmark, using different machine learning software frameworks, as well as different hardware resources. We will briefly review these results:

LIBLINEAR. In an experimental log posted in the libsvm datasets repository [18], the authors report using LIBLINEAR-CDBLOCK [19] to perform training on a single machine with 128GB of RAM. This solver performs out-of-core training by splitting the data into chunks that fit in memory. This approach takes 55 hours to train the dataset and a logarithmic loss of 0.1293 is reported.

Vowpal Wabbit. In [14], the authors evaluated the performance of Vowpal Wabbit, a fast out-of-core learning system, on the terabyte click logs dataset. Training was performed on a 12 core (24 thread) machine with 128GB of memory using Vowpal Wabbit 8.3.0 using

the first 3 billion training examples of the dataset. The training time report is approximately 333 minutes and the logarithmic loss on the test set is 0.1325.

Spark MLlib. In the same benchmark, the authors also measured the performance of the Logistic Regression provided by Spark MLlib. They deploy Spark 2.1.0 a cluster with total 512 cores and 2TB of memory. Each executor is giving 4 cores and 16TB of memory. A training time of approximately 100 minutes is reported, and the logarithmic loss on the test set is approximately 0.1320.

TensorFlow. Google have also published results where they use Google Cloud Platform to scale out the training of a Logistic Regression classifier from TensorFlow [15]. They report using 60 workers machines and 29 parameter machines for the training of the full dataset, which takes 70 minutes and produces a test loss of 0.1293.

In Figure 7, we provide a visual summary of these results. We can observe that Snap ML on 16 GPUs is capable of training such a model 46x faster than the best previously reported results (which happens to be using TensorFlow) achieving a similar level of accuracy.

5.4 Profiling

In order to understand how much of the Snap ML runtime is spent in the GPU kernel vs. how much time is spent copying data, we have performed a detailed profiling. We restrict this profiling to a smaller subset of the Terabyte Click Logs using only the first 200 million training examples and train using a single GPU. We consider two hardware configurations. Firstly, we perform the profiling using an Intel x86-based machine (Xeon** Gold 6150 CPU @ 2.70GHz) with a single NVIDIA Tesla V100 GPUs attached using the PCI Gen 3 interface. Secondly, we run the profiling on a IBM POWER AC922 server with 4 NVIDIA Tesla V100 GPUs attached using the NVLINK 2.0 interface (we only use only one of them for comparison). For the PCIe-based setup we measure an effective bandwidth of 11.8GB/s and for the NVLINK-based setup we measure an effective bandwidth of 68.1GB/s.

In Figure 8a, we show the profiling results for the x86-based setup. We can see two streams S1 and S2. On stream S1, the actual training is being performed (i.e., calls to the GPU kernel). The time to train each chunk of data is around 90ms. While the training is ongoing, in stream S2 we copy the next data chunk onto the GPU. We observe that it takes 318ms to copy the data, thus meaning that the GPU is sitting idle for quite some time and the copy time is the bottleneck. In Figure 8b, for the POWER-based setup we observe that the time to copy the next chunk is reduced significantly to 55ms (almost a factor of 6), due to the faster bandwidth provided by NVLINK 2.0. This speed-up hides the data copy time behind the kernel execution, effectively removing the copy time from the critical path and resulting in a 3.5x speed-up.

6 CONCLUSION

In this work we have described Snap Machine Learning, a new system for training of generalized linear models. This hierarchical system is built from individual components, each one implementing a state-of-the-art algorithm for parallel and distributed machine learning. This structure allows Snap ML to maximally exploit modern computing systems with multiple machines that contain both

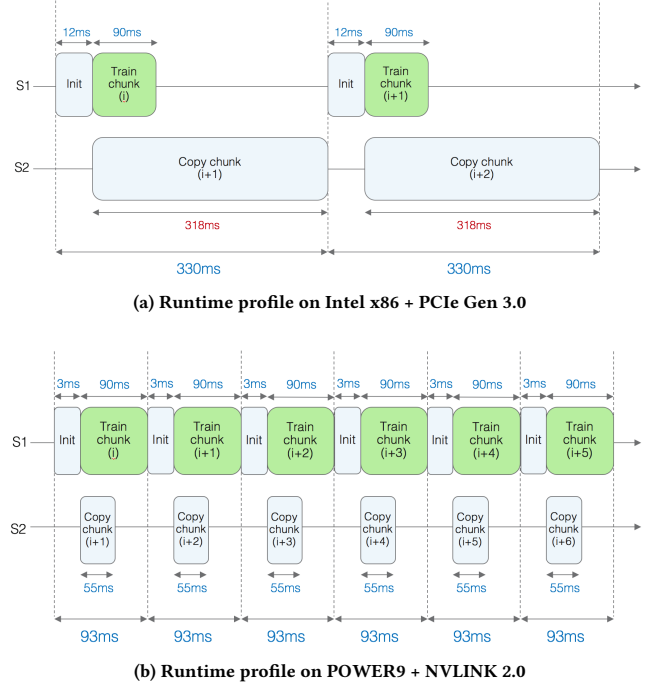


Figure 8: Runtime Profile

CPUs and GPUs, as well as high-speed interconnects and networking. We have shown that Snap ML can provide significantly faster training than existing frameworks in both single-node and multi-node benchmarks. On one of the largest publicly available datasets, we have shown that Snap ML can be used to train a logistic regression classifier in 1.5 minutes: more than an order of magnitude faster than previously reported results.

7 ACKNOWLEDGEMENT

The authors would like to thank Michael Kaufmann for testing and bug fixes, Kubilay Atasu for contributing code for load balancing, Manolis Sifalakis, Jonas Pfefferle and Urs Egger for setting up vital infrastructure. We would also like to thank Christoph Hagleitner and Cristiano Malossi for providing access to heterogenous compute resources and providing valuable support when scheduling large-scale jobs. Finally, we would also like to thank Hillery Hunter, Paul Crumley and I-Hsin Chung for providing access to the servers that were used to perform the tera-scale benchmarking.

*Trademark, service mark, registered trademark of International Business Machines Corporation in the United States, other countries, or both.

** Intel Xeon is a trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. The Apache Software Foundation (ASF) owns all Apache-related trademarks, service marks, and graphic logos on behalf of our Apache project communities, and the names of all Apache projects are trademarks of the ASF.

REFERENCES

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [2] Celestine Dünner, Thomas Parnell, Kubilay Atasu, Manolis Sifalakis, and Haris Pozidis. 2017. Understanding Optimizing Distributed Machine Learning Applications on Apache Spark. In *Proceedings of the IEEE International Conference on Big Data (IEEEBigData'17)*. Boston, MA, 99–100.
- [3] Celestine Dünner, Thomas Parnell, and Martin Jaggi. 2017. Efficient Use of Limited Memory Accelerators for Linear Learning on Heterogeneous Systems. In *Advances in Neural Information Processing Systems 30 (NIPS'17)*. Long Beach, CA, 4261–4270.
- [4] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of machine learning research* 9, Aug (2008), 1871–1874.
- [5] Message P Forum. 1994. MPI: A Message-Passing Interface Standard. *University of Tennessee* (1994).
- [6] Cho-Jui Hsieh, Hsiang-Fu Yu, and Inderjit Dhillon. 2015. PASSCoDe: Parallel ASynchronous Stochastic dual Co-ordinate Descent. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 2370–2379. <http://proceedings.mlr.press/v37/hsieha15.html>
- [7] Criteo Labs. 2015. Criteo Releases Industry's Largest-Ever Dataset for Machine Learning to Academic Community. <https://www.criteo.com/news/press-releases/2015/07/criteo-releases-industrys-largest-ever-dataset/>. (2015).
- [8] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 129–142. <https://doi.org/10.1145/1065010.1065027>
- [9] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, Feng Sun, and Hucheng Zhou. 2017. Model Ensemble for Click Prediction in Bing Search Ads. <https://www.microsoft.com/en-us/research/publication/model-ensemble-click-prediction-bing-search-ads/>
- [10] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241. <http://dl.acm.org/citation.cfm?id=2946645.2946679>
- [11] Thomas Parnell, Celestine Dünner, Kubilay Atasu, Manolis Sifalakis, and Haris Pozidis. 2017. Large-Scale Stochastic Learning Using GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS'17)*. Orlando, FL, 419–428.
- [12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [13] Virginia Smith, Simone Forte, Chenxin Ma, Martin Takáč, Michael I. Jordan, and Martin Jaggi. 2016. CoCoA: A General Framework for Communication-Efficient Distributed Optimization. *CoRR* abs/1611.02189 (2016). arXiv:1611.02189 <http://arxiv.org/abs/1611.02189>
- [14] Rambler Digital Solutions. 2017. criteo-1tb-benchmark. (2017). <https://github.com/rambler-digital-solutions/criteo-1tb-benchmark>
- [15] Andreas Sterbenz. 2017. Using Google Cloud Machine Learning to predict clicks at scale. <https://cloud.google.com/blog/big-data/2017/02/using-google-cloud-machine-learning-to-predict-clicks-at-scale>. (2017). Online; Accessed: 2018-01-25.
- [16] Andreas Töschler, Michael Jahrer, and Robert M Bell. 2009. The bigchaos solution to the netflix grand prize. (2009).
- [17] unknown. 2017. Samples for Google Cloud Machine Learning Engine. <https://github.com/GoogleCloudPlatform/cloudml-samples>. (2017).
- [18] Unknown. 2018. LIBSVM Data: Classification, Regression, and Multi-label. (2018). <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>
- [19] Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. 2012. Large linear classification when data cannot fit in memory. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 5, 4 (2012), 23.
- [20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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 (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2.