Cuckoo Linear Algebra

Li Zhou
Carnegie Mellon University
Pittsburgh, PA
lizhou@cs.cmu.edu

David G. Andersen
Carnegie Mellon University
Pittsburgh, PA
dga@cs.cmu.edu

Mu Li
Carnegie Mellon University
Pittsburgh, PA
muli@cs.cmu.edu

Alexander J. Smola
Carnegie Mellon University
Pittsburgh, PA
alex@smola.org

ABSTRACT

In this paper we present a novel data structure for sparse vectors based on Cuckoo hashing. It is highly memory efficient and allows for random access at near dense vector level rates. This allows us to solve sparse ℓ_1 programming problems *exactly* and *without preprocessing* at a cost that is identical to dense linear algebra both in terms of memory and speed. Our approach provides a feasible alternative to the hash kernel and it excels whenever exact solutions are required, such as for feature selection.

Categories and Subject Descriptors

H.1 [**Information Systems**]: Models and Principles; E.2 [**Data**]: Data Storage Representations

Keywords

Linear Models; Hashing; Sparse Vectors

1. INTRODUCTION

High dimensional sparse estimation [4] is one of the key tools in machine learning. It forms the basis of algorithms for document analysis, bioinformatics, user personalization, recommendation, and the vast array of Bayesian nonparametric models with their hierarchical and variable data structures. Unfortunately, these tools require expensive manipulation of sparse vectors. This is particularly costly whenever the sparsity pattern changes over time, e.g., by insertion of previously-unseen tokens, user actions, topics, or clusters. Usually this requires custom-built data structures to manipulate these events efficiently. In its place we propose the use of a general-purpose sparse data structure, the Cuckoo hash, as proposed by [20] with refinements due to [25, 10]. We adapt it for linear algebra and demonstrate its efficacy for sparse generalized linear models.

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.

ACM 978-1-4503-3664-2/15/08 ...$15.00.
DOI: http://dx.doi.org/10.1145/2783258.2783263.

1.1 Motivation

The motivation for our approach arises from the task of penalized risk minimization with sparsity penalty. That is, we are interested in a family of problems of the form

$$\underset{w}{\text{minimize}}\, R[w] + \lambda \|w\|_1 . \tag{1}$$

For instance, penalized logistic regression follows this template. Here $R[w]$ is the empirical risk for a given dataset $\{(x_i, y_i)\}$ with

$$R[w] = \frac{1}{m} \sum_{i=1}^{m} \log(1 + \exp(-y_i \langle w, x_i \rangle)). \tag{2}$$

Such problems are common, e.g., in spam filtering and computational advertising [5]. Similar problems arise in the context of sparse decoding where $R[w] = Pw$ for a random incoherent projection matrix P, and in the context of web page tiering [14].

One of the challenges for such high-dimensional problems is that w is often not simply a vector in $\mathbb{R}^d$ with well-defined dimensionality but rather a sparse vector of (key,value) pairs. As a result, one either needs to maintain a dictionary dynamically or it requires substantial work for preprocessing the data. In fact, the preprocessing cost can exceed the cost of solving the problem both in terms of memory footprint and in terms of computation.

To make matters more concrete, consider the case of (personalized) spam filtering [24]. In it, each document is represented as a bag of words. Hence, it is a sparse vector with an identifier for each unique word $w \in \mathcal{W}$. Moreover, for personalization purposes one computes the tensor product with the identifier associated with each user $u \in \mathcal{U}$. This leads to a space that is potentially of size $|\mathcal{W}| \times |\mathcal{U}|$, i.e., given by the product of the total number of users and words. Typical commercial e-mail systems support in the order of 100 million users and vocabularies of size 100,000 are not uncommon, thus requiring a 10^{13} dimensional space. This is clearly infeasible and unrealistic since most users use only a significantly smaller set of words and user participation follows a power law. Hence the set of unique keys is several orders of magnitude smaller.

One way of solving the associated ℓ_1 programming problem of (1) is to *preprocess* all data by mapping unique combinations of attributes and user identifiers to integers such as to form a contiguous list, i.e., to form a perfect hash. This requires sorting all entries, something that is typically accom-

plished by sorting all combinations, e.g., by using MapReduce [8]. Given m observations the cost is $O(dm \log dm)$, where d is the typical number of nonzero terms per document (we need to sort all words). Once this is accomplished, one maps all keys into integers and henceforth analysis proceeds as usual. This is the approach that LibLinear [11] and many related algorithms take. An alternative is to build a nontrivial data structure in memory to dynamically allocate storage according to the sparsity pattern, as was proposed by [21] for Latent Dirichlet Allocation. However, this is less memory-efficient than the above preprocessing step for ℓ_1 penalized models.

Unfortunately *preprocessing* is sometimes impossible with limited memory. For example when we use string kernels [13] for protein classification, the memory needed to store the expanded substring data may exceed physical memory. In this case we need a data structure that supports generating features on the fly. Also preprocessing fails whenever we are in a online setting, that is, new features keep on occurring over time. Even worse, in a distributed online setting, different features can be received by different machines and we need to have a mechanism for reconciling these subsets without the need for a common key space coordination mechanism since the latter is likely to be more costly to implement than solving the original inference problem.

In short, dealing with (key,value) pairs in machine learning is a nontrivial problem. Distributed, parallel, and online inference are substantially impeded by having to maintain a separate data structure for key management. Most importantly, in cases where the variable space already taxes the memory footprint of the system, we can ill afford sacrificing the lion's share of memory for an auxiliary data structure.

1.2 An Overview of Compact Models

An alternative way to solve the problem exactly without approximation is to store terms compactly, or to employ domain specific variable compression. Below we review three major approaches and discuss their relative merit.

Hash Kernels: This is the most obvious alternative to Cuckoo hashing. In fact, if we only care about predictive accuracy rather than the ability to select attributes and debug the model, then this is a credible alternative [24]: instead of computing linear functions via

$$f(x) = \sum_i w_i x_i \tag{3}$$

one aims to solve the problem via

$$f_{\text{hash}}(x) = \sum_i w[h(i)]\sigma(i)x_i. \tag{4}$$

Here $h(i)$ is a hash function mapping the set of keys i to some range $\{1 \dots N\}$ and moreover, $\sigma(i)$ is a binary Rademacher hash mapping keys with equal probability to $\{\pm 1\}$. One may show that in expectation this construction preserves inner products. Moreover, the memory footprint is automatically limited to N floating point locations.

Given N storage locations and a set of $n+1$ keys it follows that the probability of a collision is given by

$$1 - \left(1 - \frac{1}{N}\right)^n > 1 - e^{-\frac{n}{N}} \tag{5}$$

That is, the probability for a given memory location to be free is $(1 - 1/N)$. Hence after n insertions it is $(1 - 1/N)^n$. Hence, even if we have sufficient memory to store the keys when preprocessing the data explicitly, i.e., $n = N$, it follows that the collision probability is still $1 - e^{-1} > 0.63$. In other words, approximately 37% of the memory remains unused for storing data.

Moreover, many of the guarantees are specific to *inner products*. That is, unless we use this only for computing $\langle w, x \rangle$ rather than, say, nonlinear function classes, the approximation guarantees may not even be valid.

Conditional Random Sampling is an attractive strategy for computation reduction in dealing with textual data. It was proposed in [16] to sparsify sparse vectors even further by subsampling entries efficiently. In a nutshell, it works as follows: similar to minwise hashing [6] it applies a random permutation to the key space and retains the k smallest keys. However, it uses not only the keys but also their permutation rank to control for the cardinality of the set.

This strategy limits the memory footprint of the *data* x rather than of the *parameter vector* w. Hence we can view it as a complementary rather than competing technique to what we aim to accomplish — a compact and dynamically adjustable data structure for linear models.

Suffix Trees: When dealing with string kernels it is possible to gain additional efficiencies by storing documents in the form of suffix trees [22, 23]. Denote by s a string and by $s[i:j]$ its substring starting at position i and ending at position j. In this case, parsing all *training data* into a joint suffix tree allows one to compute a kernel expansion using weighted substrings. That is, given weights $\lambda_{x[i:i+\tau]} \geq 0$ for substrings we compute

$$k(x, x') = \sum_{i,j,\tau} \lambda_{x[i:i+\tau]} \delta(x[i:i+\tau], x'[j:j+\tau]) \tag{6}$$

in $O(|x|+|x'|)$ time. Moreover we can evaluate any linear combination of such attributes $f(x) = \sum_{i,\tau} w_{x[i:i+\tau]}$ in $O(|x|)$ time. This is very attractive, provided that the training set is sufficiently small.

Unfortunately, even the most lightweight implementation of suffix trees still has a memory footprint of up to 40 bytes per character, plus storage for the actual parameter payload. Assuming single precision floating point numbers this amounts to an overhead of 1000%, i.e., less than one tenth of the space is used to store the actual model. Moreover, it is challenging to select good subsets of attributes. Hence, even for documents, specialized string kernels are insufficient to address the problem of high dimensionality.

1.3 Our Contribution

In this paper we propose an alternative to the above strategies that uses Cuckoo hashing as the underlying data structure. Its benefits are as follows:

- The data structure requires *no preprocessing* and it can adapt to new attributes on the fly.

Method	Preprocessing	Accuracy	Memory	Speed	Incremental Data
Dense Array	required	exact	normal	fast	no
C++ STL Hash	no	exact	high	slow	support
Hash Kernel	no	inexact	low	fast	support
Cuckoo	no	exact	normal	fast	support

Table 1: Comparison between four methods of dealing with sparse vector in machine learning algorithms.

- It is convenient to use the data structure in various sparse linear machine learning algorithms, and achieve a high speed that is close to dense vector structure.
- The data structure is essentially exact. That is, given n distinct keys, we only need an overhead of $2 + \log_2 n$ bits per memory location to ensure that no collision occurs, hence the model recovers all attributes exactly. Moreover, the expected number of collisions increases gracefully as the metadata decreases.
- The occupancy ratio of the data structure is high, and can sometimes higher than 90%. In practice we find that LibLinear [11] with Cuckoo hashing only uses 10% more memory than with dense vectors.
- The data structure can be used easily in a distributed optimization setting since it requires *no coordination* in terms of key space between encodings on different machines, even if they receive different keys.
- The data structure can handle novel keys arriving at runtime, e.g., in an online learning setting, without the need for maintaining and storing a tokenization dictionary.
- The data structure is *size adaptive*. That is, its memory footprint scales (and resizes) almost linearly with the number of keys required.

These desirable properties allow us to solve ℓ_1 penalized optimization problems and similar sparsity constrained problems without preprocessing or approximation, and makes the data structure an important component for distributed (online) inference algorithms on sparse vectorial data.

Furthermore, in Table 1 we compare Cuckoo with other methods in various aspects such speed, memory, accuracy and abilities.

2. CUCKOO HASHING

To serve as a self-contained description, we briefly overview the theory and practice of Cuckoo hashing. Since its first description by Pagh et al. [20], it has received considerable attention both in theory and practice. At its heart, Cuckoo hashing relies on on the "power of two choices": when performing randomized load balancing, being able to choose between two (nearly random) buckets dramatically reduces the maximum load on any single bucket.

2.1 Data Structure

Our work uses the recently proposed "partial-key" Cuckoo variant [25], which is fast and achieves high (over 90%) table occupancy. We describe the data structure in two steps: first by explaining basic associative Cuckoo hashing, and then by explaining the partial-key variant.

Like all hash tables, a Cuckoo hash table consists of an array of memory broken into "buckets." A hash function H maps a key to be looked up or inserted to to a number in the range $(0, |\text{buckets}|)$.

In the "2,4-Cuckoo" variant of Cuckoo hashing, *two* hash functions are applied to keys, producing h_1 and h_2, and each bucket contains 4 "slots" for individual key/value items. Each slot can contain any item that maps to that bucket. The process of searching for an item in a Cuckoo hash table is simple: `Get(key)`: Compute h_1 and h_2, retrieve buckets b_1 and b_2, and examine the $2 \times 4 = 8$ potential items found in them. If the key stored in the slot matches the key being searched for, return the associated value.

The theoretical challenge of Cuckoo hashing is on item insertion: For key k_{insert}, if all slots in buckets b_1 and b_2 are full, then insert selects an item, k_{victim} in one of these buckets for displacement. It inserts k_{insert} in its place. It then re-inserts k_{victim} into the alternate bucket for k_{victim}. If that alternate bucket is full, the process recurses, displacing yet another key, and so on, until the items have been successfully inserted. If the process does not succeed within several hundred displacements, the table is full, and a resize procedure is invoked. The theory of Cuckoo hashing indicates that this process works with high probability, allows extremely high table occupancy, and has amortized $O(1)$ time to insert (though some individual insertions may be slow).

The *partial-key* variant permits the table to not store the full key in the slot. Instead, the table stores only some number of bits of the (hashed) key. Cuckooing works as normal. Retrieval, however, may experience *false positives*: A slot may appear to contain the desired key when it does not. For a 2,4 Cuckoo table with 8 possible slots for each key, a false positive can occur with probability $\frac{8}{2^{\text{nbits}}}$, where nbits is the number of key bits stored in the table.

- We denote by $x : \mathcal{I} \to \mathbb{R}$ the sparse vector referenced by the index set $\mathcal{I}$.
- $s : \mathcal{I} \to \{0, 2^d - 1\}$ is the (hash) signature of i
- $h : \mathcal{I} \to \{0, 2^b - 1\}$ is the hash function mapping the index set into a given range of values given by the hash vector.
- $g := h \circ s$ is the hash function mapping between the alternative locations for a key. Whenever appropriate we will refer to $h_1 := h$ and $h_2 := h\,\text{XOR}(h \circ s)$ as the alternate hash functions. By construction $h_1 = h_2\,\text{XOR}(h \circ s)$.
- With some abuse of notation the storage itself is denoted by $x \in \left(\mathbb{R}, \{0, 2^d - 1\}\right)^{2^b}$ consisting of tuples of values and signatures of the keys.

Optimizations for Locality.

Using this mechanism as a building block, we face the challenge of efficiently implementing sparse vector operations atop it. A naive use of hash tables as a sparse vector representation requires excessive amounts of random access when, e.g., computing the products of vectors or adding them together: Because the same key may be stored at different locations in each hash table, one cannot simply sequentially traverse both data structures.

In nearly all vector-vector operations, at least one of the two vectors can be traversed sequentially (i.e., in random key order, but sequentially through memory). Such a traversal is fast: modern CPUs have extremely high sequential memory bandwidth. However, if the values of interest are not stored in the same order in the second vector (because of cuckooing, or because they are of different sizes), these values need to be accessed randomly.

We describe below our algorithms for sparse vector operations using partial-key Cuckoo hashing. These algorithms take advantage of two optimizations:

Prefetching: When traversing a vector sequentially, create a "pipeline" of k items: $x_1, x_2, x_3, \ldots, x_k$. Issue memory prefetches for items $1, \ldots, k$ before returning to item $x_1, \ldots, x_n$ to perform the actual operation. While these are still random memory fetches, this strategy hides the latency of access to memory, and substantially boosts performance.

First-Biased Insertion: In our Cuckoo hash tables, the insertion procedure defaults to inserting item x at its first hash, h_1, if a slot is available there. By leaving the table somewhat more sparse than is required, and by using wider associativity (e.g., 2,8-Cuckoo), more items can be inserted into their first-hash location. This will induce a larger similarity of memory locality between two same-sized sparse vectors with many shared items, which can modestly speed the throughput of, e.g., vector addition.

2.2 Linear Algebra

We now adapt the basic data structure to perform efficient linear algebra. Our primary focus is on probabilistic Cuckoo hashing: representations that have a small but nonzero probability of collision. The operations of primary interest are inner products $\langle w, x \rangle$, vector updates $w \leftarrow w + \delta$, and traversal of the nonzero elements of a vector; these are the tools needed to design linear models. In linear algebra these are referred to as Level 1 BLAS subroutines [9]. For simplicity of exposition, we omit from the description the handling of the (typically four) associative slots per hash bucket. Their handling is straightforward—simply checking four locations at a time—and the performance reduction is minimal, because the slots map to the same CPU cache line. (Four or eight-way associativity is key to achieving high occupancy.)

2.2.1 *Element-wise operations*

To compute the $\|x\|_p^p$ norm we must scan all entries $z[j]$. Because summation is commutative, the scan can traverse x in the order the entries are stored in the Cuckoo hash table:

```
norm ← 0
for j = 0 to 2^d − 1 do
  norm ← norm + |z[j].value|^p
end for
```

Likewise, finding the largest element is order-invariant. Hence a single pass in the order in which elements are stored in x suffices.

Cuckoo hashes have the attractive property that *any* pointwise operation can be carried out by a simple scan, i.e., in $O(n)$ time, given n nonzero entries. This is useful, e.g., when we want to apply the prox operator in algorithms such as FISTA [3]. There one applies a gradient element-wise to the entries and subsequently all entries are shrunk back towards 0, hence the name of the algorithm (Fast Iterative Successive Thresholding).

2.2.2 *Dot product* $\langle x, y \rangle$

Computing the dot product requires iterating over all matching nonzero entries of x and y. It suffices to check, for each non-zero entry in the vector with the fewest entries, whether that entry matches in the more dense vector. Without loss of generality, assume that x is more sparse. This algorithm is the first that uses biased insertion to optimize its retrieval: When examining item j in vector x, it first tests the equivalent slot j in vector y. These two retrievals are both a sequential traversal of their respective vectors. If the item is not found in its primary location, the algorithm will search its alternate location in y, which requires a random access:

```
dot ← 0
for j = 0 with |x[j].value ≠ 0| to 2^d − 1 do
  if x[j].signature = y[j].signature then
    dot ← dot + x[j].value · y[j].value
  else
    h2 ← h(x[j].signature) XOR j
    if x[j].signature = y[h2].signature then
      dot ← dot + x[j].value · y[h2].value
    end if
  end if
end for
```

There is obvious benefit in prefetching whenever the signatures do not match, i.e., whenever

$$x[j].\text{signature} \neq y[j].\text{signature}$$

and whenever x is a shorter vector than y even for

$$x[j].\text{signature} = y[j].\text{signature}.$$

Hence we look up keys in a non-blocking fashion by issuing a prefetch request, carry on with the current workload and return to the queued-up terms once they are available. As before, the total cost is $O(\min(n_x, n_y))$ since the procedure must iterate only over the smaller of the two vectors.

2.2.3 *Vector addition* $z = ax + y$

Vector addition is analogous to the dot product, but it cannot skip non-zero entries in either vector, and requires a scalar multiplication of every element in x by a.

If y has fewer elements than x, it is best to accomplish the addition in a separate step. First, set $z \leftarrow x$, and multiply it by a:

```
for j = 0 with |z[j].value ≠ 0| to 2^d − 1 do
  z[j].value *= a
end for
```

And then traverse y (the sparse vector), and add its elements into z:

```
for j = 0 with |y[j].value ≠ 0| to 2^d − 1 do
  if z[j].signature = y[j].signature then
    z[j].value += y[j].value
  else
    h2 ← h(y[j].signature) XOR j
```

```
    if z[h2].signature = y[j].signature then
      z[h2].value += y[j].value
    end if
  end if
end for
```

The case where x is more sparse than y is similar, except that the multiplication and addition can be merged. z is instead initialized as a copy of y, and the non-zero elements in x are multiplied by a and pushed into z. While a copy of a sparse Cuckoo vector is fast (a sequential bulk memcpy), if the destination vector is one of the source vectors, the copy step can be omitted, as is done in the BLAS SAXPY operation.

As before, this can be carried out in linear time, albeit this time in $O(n_x + \min(n_y, n_y))$ operations due to the pointwise multiplication $x \leftarrow ax$. For stochastic gradient descent procedures, the number of nonzeros in x is low since we need to implement $w \leftarrow w - \eta_t g_t$ where g_t is the gradient computed e.g., on a given document. As a result the updates are essentially as fast as performing $O(1)$ random access to the parameter vector.

3. MACHINE LEARNING ALGORITHMS

The previous two sections motivated the need for a new data structure and showed how it could be constructed efficiently. To demonstrate its versatility, we now apply it to three representative machine learning settings: batch inference, online inference, and distributed inference.

3.1 Batch Inference

We use LibLinear [11], a popular machine learning package for solving sparse linear optimization problems, as the baseline for comparison. In a nutshell, LibLinear solves the optimization problem (1) with logistic loss function (2) as risk $R[w]$. The gradient and Hessian of the risk $R[w]$ can be written as

$$\partial_w R[w] = \frac{1}{m}\sum_{i=1}^{m} \frac{-y_i x_i}{1+\exp(y_i \langle w, x_i\rangle)}$$

$$\partial_w^2 R[w] = \frac{1}{m}\sum_{i=1}^{m} \frac{x_i x_i^\top}{\left(\exp\left(\frac{1}{2}\langle w, x_i\rangle\right) + \exp\left(-\frac{1}{2}\langle w, x_i\rangle\right)\right)^2}$$

Here we use only the diagonal component of the Hessian $\partial_w^2 R[w]$ as a preconditioner. This yields the search direction

$$g = \left(\mathrm{diag}\partial_w^2 R[w]\right)^{-1} \partial_w R[w]. \tag{7}$$

which is used for line search via

$$\begin{aligned} &R[w+\beta g] + \lambda \|w+\beta g\|_1 - R[w] - \lambda \|w\|_1 \\ \leq &\beta \langle g, \partial_w R[w]\rangle + \lambda[\|w+\beta g\|_1 - \|w\|_1]. \end{aligned}$$

Here g is the search direction, $\beta \in (0,1)$ is the step size and λ is the regularization constant.

LibLinear implements an improved GLMNET algorithm analogous to what was summarized above: in its implementation all features are preprocessed and indexed by integers for direct memory access. C-style arrays are used to store features related data such as weight w and gradient $\partial_w R[w]$.

Extending LibLinear to Cuckoo hashing is straightforward: we only need to substitute all feature related C-style arrays with Cuckoo hash tables. In doing so, the algorithm can use the original training data *directly* as input.

3.2 Online Inference

Cuckoo hashing excels particularly when processing variable feature sets in an online fashion. Moreover, it is very amenable to element-wise operations. Hence we implemented the follow-the-regularized-leader proximal [18] algorithm (FTRL-Proximal) using Cuckoo hashing and hash kernels as alternative strategies.

FTRL-Proximal algorithm is a sparse online algorithm. It proceeds as follows: given a sequence of gradients, $g_t \in R^d$, it performs the update

$$w_{t+1} = \operatorname*{argmin}_w \left[\langle g_t, w\rangle + \frac{1}{2}\sum_{s=1}^{t}\sigma_s \|w - w_s\|_2^2 + \lambda \|w\|_1\right]$$

where σ_s is the learning-rate. Using $\lambda > 0$, FTRL-Proximal introduces excellent sparsity. If we store $z_{t-1} = g_{1:t-1} - \sum_{s=1}^{t-1}\sigma_s w_s$, and let $z_t = z_{t-1} + g_t + (\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}})w_t$, then w_{t+1} is solved in the following closed form on a per-coordinate bases [19]

$$w_{t+1,i} = \begin{cases} 0 & \text{if } |z_{t,i}| \leq \lambda \\ -\eta_t(z_{t,i} - sgn(z_{t,i})\lambda) & \text{otherwise} \end{cases}$$

Note that this operation can be carried out in the Cuckoo hash space since the problem decomposes into d scalar optimization problems. Hence it can be implemented very efficiently.

3.3 Distributed Inference

Distributed optimization and inference is a prerequisite for solving large scale machine learning problems. There are several open source distributed machine learning frameworks available, such as Parameter Server [15], YahooLDA [1], GraphLab [17], Petuum[7], Mahout [2], and MLBase [12]. We use the Parameter Server as the motivating example, but the discussion can be extends to other frameworks as well.

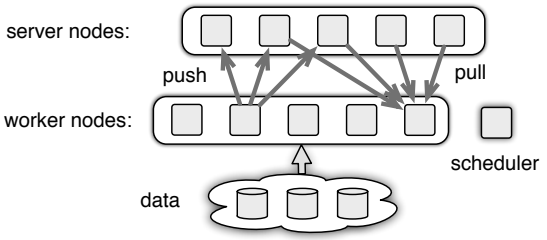

Figure 1: A simplified architecture of parameter server [15]

In parameter server, there are three different nodes, which are shown in Figure 1. The globally shared parameters w are partitioned and stored in the server nodes. Each worker node solves a subproblem of $R[w]$ and communicates with the server nodes in two ways: to push local results such as gradients or parameter updates to the servers, and to pull recent parameter (changes) from the servers.

In the distributed environment, preprocessing data locally in worker node is impossible, because we may assign different local indices to the same feature in different work nodes, and then we can not match them in the server node. Global data preprocessing, however, requires extensive data communication, which could be a bottleneck of the system.

	Dense Array	Cuckoo	Hash Kernel ($N = 27$)	Hash Kernel ($N = 20$)
Preprocessing (sec)	514	0	0	0
Data Transformation (sec)	32	122	70	50
Training Time (sec)	2077	1520	2373	370
Total Time (sec)	2623	1643	2444	421
Memory Used (GB)	23	23	30	20
Accuracy	93.22%	93.23%	93.25%	90.96%
Feature Reconstruction	0.8777	0.8773	0.2638	0.0052

Table 2: Comparison of dense array, Cuckoo, and hash kernel with bit length $N = 20$ and $N = 27$ on LibLinear with CTR dataset. The best results are colored by Red and the second best by Green.

To implement the FTRL-Proximal algorithm discussed in the previous section on parameter server, we implemented a Cuckoo hashing based server nodes to handle feature keys pushed by the worker nodes. Therefore we can get rid of the expensive global preprocessing.

4. EXPERIMENTS

We use the sparse logistic regression discussed in Section 1.1 as the benchmark algorithm. We evaluate Cuckoo linear algebra in the following three settings discussed in Section 3: batch inference, online inference, and distributed inference. The Cuckoo linear algebra implementation is based on libcuckoo library[1].

To evaluate the performance, we use two binary classification datasets. The first one is a DNA dataset from the Pascal Large Scale Learning Challenge[2]. Each instance in the dataset is represented by an ASCII string of length 200 with symbols {A, C, G, T}. There are 50 million training instances and 1 million validation instances. As the labels are extremely unbalanced (a positive/negative ratio of 0.0028), we use Area Under the Curve (AUC) as our metric.

The second dataset is an anonymous online advertising click-through rate (CTR) dataset. In this dataset each instance is a display of an ad, and the label is 1 if it is clicked by a user and 0 otherwise. For single machine experiments we sampled about 8.3 million instances and 100 million features, and for distributed experiments we sampled about 20 million instances and 200 million features. Each feature is represented by a 64-bit signature. Since this dataset cannot be used by LibLinear directly, we first do preprocessing to map feature signatures to continuous indices. Note, however, this dataset can be used directly by Cuckoo linear algebra and hash kernel.

All experiments were carried on a university cluster. Each machine is equipped with one Intel Xeon E5620 2.4GHz CPU, 64GB memory, and 1 Gigabyte Ethernet. All compared methods are implemented by C/C++ and compiled with GCC 4.8.

4.1 Batch Inference

Setup. In this experiment we compare the following three methods.

LibLinear: LibLinear uses a dense array implementation. Consequently, before invoking LibLinear, the data must be preprocessed. We wrote an additional preprocessing function to map the tokens or the 64-bit signatures of the features to continuous integer indices and build the mapping dictionary.

Cuckoo: We substituted all feature related dense arrays in LibLinear with Cuckoo hash tables. To keep IO times comparable we represent feature keys by 64-bit signatures.

Hash Kernels: As a second alternative, we implemented a hash kernel for LibLinear. That is, we hash feature keys at random into bins as described in Eq. (4): Hashing the 64-bit keys to b bits when reading training data from disk, multiplying them with a random (Rademacher) hash. In other words, we hash into an $N = 2^b$ dimensional space. After that we invoke LibLinear's ℓ_1 logistic regression procedure to train the model.

We evaluated these three methods on the CTR dataset. For each method, we report the time, memory consumption, and feature reconstruction. The feature reconstruction is measured by the Jaccard similarity. In other words, we first obtained a baseline model from LibLinear, then we ran each method several times and compared the nonzero entries between the baseline model and the model of each method. Denote the nonzero feature sets of two model by A and B, the Jaccard similarity between these two model is

$$JS(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{8}$$

We ran each method 10 times and reported the average Jaccard similarity. The results are shown in Table 2 [3].

Speed. We first observe that the preprocessing step for LibLinear with dense array takes about 500 seconds, which is roughly 20% of the total running time. In the preprocessing step we need to read all the training data, construct a feature vocabulary, map features to continuous indices and then write the whole training data to the disk. Both mapping and disk IO are time consuming.

The next step is data transformation, which transforms the sparse training matrix from row-major format into column-major format. Taking the advantage of continuous indices, LibLinear with dense array uses the least time.

In the following training part, it is quite surprising that Cuckoo outperforms the dense array implementation. The reason are twofold: first, the time complexity for accessing

[1] https://github.com/efficient/libcuckoo

[2] http://largescale.ml.tu-berlin.de/

[3] In the process of our evaluation, we discovered and fixed several memory leaks in LibLinear. As a result, the version of LibLinear we use for evaluation uses, without any algorithmic changes, 30% less memory than the originally-released version [11].

Maximum length of substring = 16			
Method	AUC	Time (sec)	Memory (GB)
Cuckoo	0.7671	32259.9	30
STL Hash	0.7671	63201.0	46.5
Hash Kernel (N=27)	0.7604	10923.9	3
Dense Array	-	-	460*
Maximum length of substring = 14			
Method	AUC	Time (sec)	Memory (GB)
Cuckoo	0.7668	26173.1	7
STL Hash	0.7668	46363.1	12
Hash Kernel (N=27)	0.7554	9145.5	3
Dense Array	-	-	460*

Table 3: Comparison of all methods on the online FTRL algorithm with DNA dataset. (*estimated values).

an element in Cuckoo hash table is $O(1)$, and with well optimized technologies such as prefetching as we mentioned before, the performance of Cuckoo hashing is close to dense array. Second, the structural optimization used by Cuckoo hash table induces an more compact representation of sparse vectors and therefore improves the sparse locality. A similar results can be observed from hash kernel: A small bit length ($N = 20$) is almost 5 times faster than a large bit length ($N = 27$), since the 20-bit hash kernel fits into L3 cache and has better sparse locality.

Memory. In terms of memory, Cuckoo used about the same memory as LibLinear. The reason is that Cuckoo hash table can have an occupancy ratio greater than 90%, so its memory consumption is almost the same as a dense array. On the other hand, hash kernel with a small bit length can effectively compress the weight and therefore used less memory than others.

Accuracy. When comparing model performance, LibLinear, Cuckoo and hash kernel ($N = 27$) provided nearly the same accuracy. This is reasonable since LibLinear and Cuckoo linear algebra both provide exact solution, and when $N = 27$, hash kernel's key space is slightly larger than the number of features in the dataset (100 million, or about $2^{26.5}$). Unfortunately, with this accuracy, hash kernel was slower than Cuckoo and consumed more memory than both Cuckoo and dense array. For $N = 20$, hash kernel was the fastest within the four methods, took only 421 seconds, and consumed only 23 GB memory. However, the accuracy dropped significantly to 90.96%.

Feature Reconstruction. One very useful attribute of $\ell 1$ regularizer is feature selection, therefore feature reconstruction is also an important merit. The reconstruction rate of LibLinear is 0.8777, which is less than 1 because the $\ell 1$ regularizer does not guarantee unique solution and the inference algorithm (GLMNET) is a randomized method. The reconstruction rate of Cuckoo is 0.8773, which is nearly identical to LinLinear, indicating that Cuckoo has perfect feature reconstruction ability. The results of hash kernel is 0.2638 for $N = 27$ and 0.0052 for $N = 20$, which is significantly worse. Note that even though 27-bit hash kernel used more memory than dense array, the unavoidable conflict destroys the reconstruction.

In summary, Cuckoo linear algebra is comparable with the dense array in terms of memory consumption, accuracy, and feature reconstruction while it is faster than the latter. Hash kernel, on the other hand, can trade off between the memory/speed and accuracy, however it cannot win on both sides. In addition, it sacrifices the interpretability.

4.2 Online Inference

Setup. In the online inference experiment, we generated the features on the fly. In particular, we used the DNA dataset and chose the substrings of the original DNA strings as features. In keeping with substring kernels of [13] we instantiated all substrings of length 1 to 16, that is, 3,080 substrings per string. On average each string has 2,477 distinct substrings, and the total number of distinct substrings in the dataset is about 800 million. Each of these terms was weighted exponentially according to its length

$$s \to (\ldots \gamma^{\tau} s[i : i + \tau] \ldots) \text{ for } \gamma = 0.95. \tag{9}$$

In other words, long strings are penalized further.

Using the DNA dataset we simulated the situation where there are a huge number of potential features and generating a vocabulary for preprocessing would be infeasible. On the other hand, both Cuckoo hashing and hash kernel can support generating features during the training phase without a vocabulary.

Figure 2 shows that the number of unique features increases exponentially with the maximum substring length. With maximum length equals to 16, 800 million features are generated.

Besides Cuckoo linear algebra and hash kernel, we added an implementation with C++ STL hash for reference. Figure 3 shows the memory footprint required to store all the features as the maximum substring length increases from 1 to 16. As we can see, with 800 million features, dense array consumed only 12 GB memory, which is the least among the three. This is reasonable since dense array does not have any overhead storing these features. With the benefit of high occupancy ratio, Cuckoo hash table consumed only 18 GB memory while STL hash table consumed more than 32 GB memory. Also, on average Cuckoo hash table consumed half as much as the memory used by STL Hash as the number of feature increased.

In our implementation of Cuckoo hash table, we allocate the slots as power of 2, so if the number of features exceed 2^k, we reallocate 2^{k+1} slots. On average Cuckoo hash table consumed 1.7 times more memory than dense array, however if the number of features is less than and close to 2^k, then the occupancy ratio of Cuckoo hash table is high and the memory usage is close to dense array. For example, when the number of feature is about 450 million, Cuckoo hash table

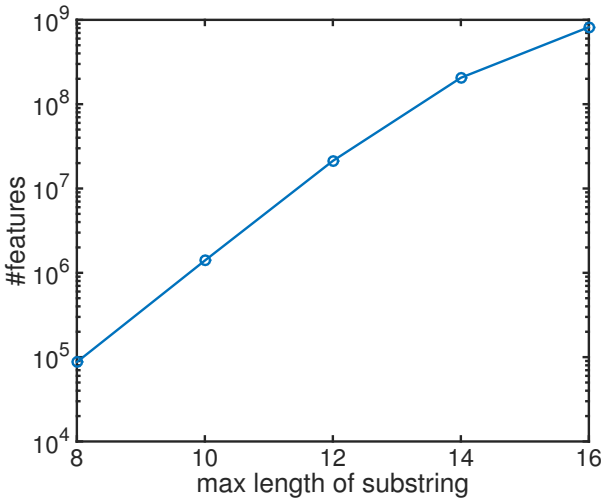

Figure 2: The number of unique features versus the maximum length of substring on dataset DNA.

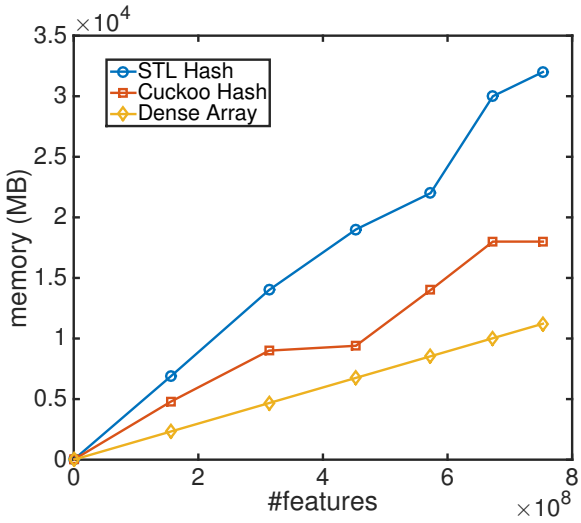

Figure 3: Comparison of the memory usage among all methods by different the number of features.

consumed about 9 GB memory and dense array consumed about 6.8 GB memory, while STL hash table consumed 2 times more.

Varying maximal substring length. We compared these methods by varying the maximum substring length. The results when maximum length equals to 14 and 16 are shown in Table 3. As can be seen, preprocessing on this dataset is impossible: on average each instance has 2,477 features, and assume each feature is encoded as a 32-bit integer, then it takes about 460 GB to store all these 50 million training instances, so it cannot be fitted into the memory and trained with LibLinear.

The accuracy of Cuckoo and STL Hash are similar, however, Cuckoo uses 40% less memory and is 1.7 times faster than the STL hash implementation. Hash kernel can reduce the memory footprint and time further, however it loses accuracy.

Limited memory or time. We evaluated the accuracy of these methods under maximum allowable memory constraints by varying the maximum substring length. Figure 4 shows the results. Given the same memory constraint, Cuckoo was more memory efficient and can store more features than STL hash, therefore it obtained higher accuracy. We also varied the bit length of the hash kernel, and reported the best results on different substring length. Using small bit length, the superiority of hash kernel allows it to process more features and therefore was comparable with the STL hash. However, when using a larger bit length it performed worse.

Next, we evaluated these methods under maximum allowable time constraints with substring length fixed to 16. The results is shown in Figure 5. Cuckoo outperformed STL hash as it is faster and able to process more examples. It is worth noting that hash kernel is comparable with Cuckoo, the reason is that hash kernel also has a constant time complexity and thus can process more examples in the given time to compensate the loss due to feature conflict.

In summary, both Cuckoo, STL hash and hash kernel support generating features on the fly and thus online inference.

	STL Hash	Cuckoo	Hash Kernel
AUC	0.6705	0.6705	0.6682

Table 4: AUC of distributed FTRL-Proximal algorithm with 6 server nodes and 12 worker nodes and CTR dataset.

However Cuckoo is more CPU and memory efficient than STL hash and it provides exact solution, thus under a fixed memory constraint or time constraint greater than 1600 seconds, Cuckoo outperformed both STL Hash and hash kernel and achieved the highest AUC.

4.3 Distributed Inference

Setup. We use the parameter server [15] as our machine learning framework to examine Cuckoo linear algebra in the distributed inference. We directly modified the built-in implement of FTRL-Proximal algorithm in parameter server. In other words, we substituted the STL hash used by the server node with Cuckoo linear algebra and hash kernel in our experiments.

We fixed the number of machines to 6, and each machine has 2 workers. This configuration can fully utilize the computational power. We varied the number of server nodes between 1 and 12. Decreasing the number of server nodes will increase the workload of each server. For hash kernel, we fix the bit length to 24.

Vary the number of server nodes. Experiment results are shown in Figure 6. Compared with STL hash implementation, Cuckoo reduced the total training time of the whole system by more than 30% when the server nodes are more than 2. The significant performance boost also indicates that the server nodes are the bottleneck of the parameter server. In terms of memory, Cuckoo used 25% less memory than STL hash when the number of server is 3 or 6, and with Cuckoo we can serve all workers with only 1 or 2 server nodes, while the STL hash implementation ran out of memory.

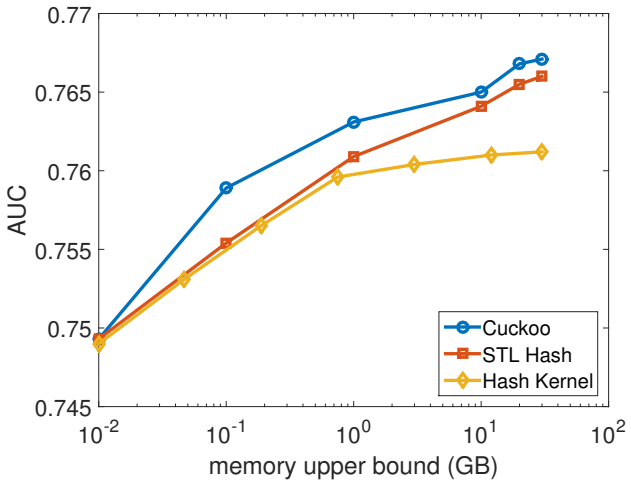

Figure 4: AUC under different memory constraints by varying the maximum length of substring from 1 to 16. We also varied the bit length of hash kernel and reported the best results on all substring length.

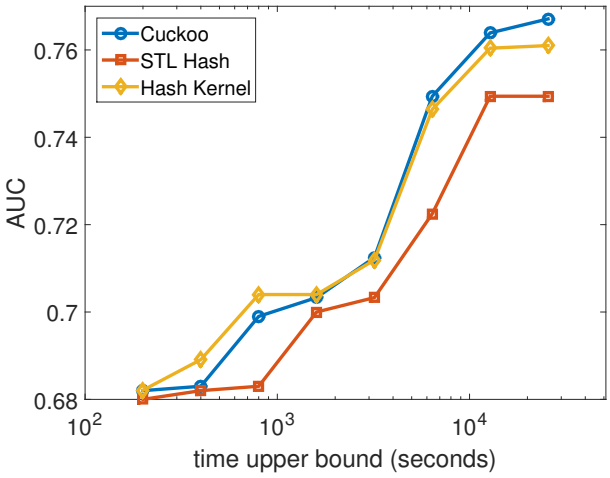

Figure 5: AUC under different time constraints by fixing the maximum length of substring to 16.

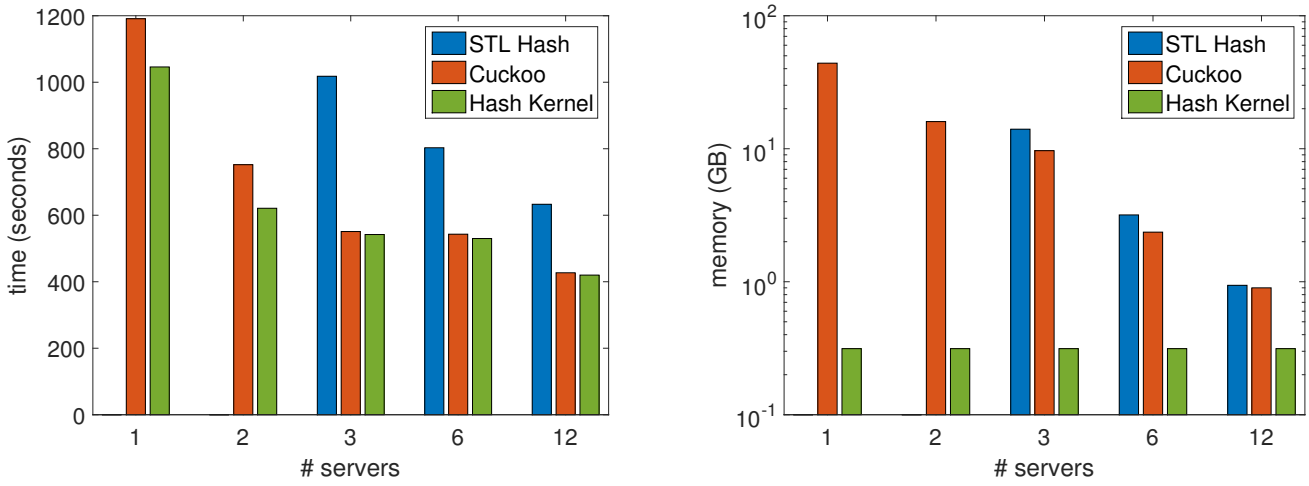

Figure 6: Comparison of time and memory by varying the number of server nodes in the parameter server. Note that when the number of servers is 1 or 2, the STL hash implementation ran out of memory.

Hash kernel, on the other handle, has similar speed as Cuckoo. It can further reduce the memory requirement as expected. However, it decreased the accuracy. Table 4 shows the AUC achieved by these three methods. As can be seen, Cuckoo is comparable with STL hash, and outperforms hash kernel by .2% AUC. Due to the commercial importance of ads click prediction, even an .1% improvement brings significant revenue improvement for the whole company.

As a summary, Cuckoo linear algebra is a perfect substitute for hash table used by distributed machine learning frameworks to improve both CPU and memory efficiency.

5. CONCLUSION

In this paper we proposed to use Cuckoo hash as the underlying data structure for sparse vectors. It is highly memory efficient and allows for random access at near dense vector level rates. We defined linear algebra operations based on this representation, and showed it can be easily applied to various machine learning algorithms, including batch inference, online inference, and distributed inference. We demonstrated its efficacy by comparing with commonly used methods, such as dense array with data preprocessing, standard STL hash, and hash kernel with different bit length. The experimental results showed that Cuckoo linear algebra outperform the others either in speed/memory, accuracy or both.

Acknowledgments.
The authors thank Paul Bradley, Tyler Johnson, and Carlos Guestrin for inspiring discussions and experimental investigation in the context of hash kernels and sparsity and

Quoc Le regarding optimization. Parts of this work were supported by grants from Google and Amazon.

6. REFERENCES

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[2] Apache Foundation. Mahout project, 2012. http://mahout.apache.org.

[3] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[4] P. Bickel, Y. Ritov, and A. Tsybakov. Simultaneous analysis of lasso and dantzig selector. *Annals of Statistics*, 37(4), 2008. Comment: Noramlization factor corrected.

[5] A. Z. Broder. Computational advertising and recommender systems. In P. Pu, D. G. Bridge, B. Mobasher, and F. Ricci, editors, *Conference on Recommender Systems, RecSys*, pages 1–2. ACM, 2008.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[7] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2004.

[9] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[10] B. Fan, D. G. Andersen, and M. Kaminsky. The cuckoo filter: ItâĂŹs better than bloom. *USENIX ;login:*, 2013.

[11] R.-E. Fan, J.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, Aug. 2008.

[12] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[13] C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 564–575, Singapore, 2002. World Scientific Publishing.

[14] G. Leung, N. Quadrianto, A. J. Smola, and K. Tsioutsiouliklis. Optimal web-scale tiering as a flow problem. In *NIPS*, pages 1333–1341, 2010.

[15] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Amhed, V. Josifovski, J. Long, E. Shekita, and B. Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[16] P. Li, K. Church, and T. Hastie. Conditional random sampling: A sketch-based sampling technique for sparse data. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 873–880. MIT Press, Cambridge, MA, 2007.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[18] B. McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization. In *International Conference on Artificial Intelligence and Statistics*, pages 525–533, 2011.

[19] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, and D. Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[21] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[22] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[23] S. V. N. Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 569–576. MIT Press, Cambridge, MA, 2003.

[24] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. J. Smola. Feature hashing for large scale multitask learning. In L. Bottou and M. Littman, editors, *International Conference on Machine Learning*, 2009.

[25] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.