Weightless: Lossy weight encoding for deep neural network compression

Brandon Reagen [1] **Udit Gupta** [1] **Robert Adolf** [1] **Michael M. Mitzenmacher** [1] **Alexander M. Rush** [1]
Gu-Yeon Wei [1] **David Brooks** [1 2]

Abstract

The large memory requirements of deep neural networks limit their deployment and adoption on many devices. Model compression methods effectively reduce the memory requirements of these models, usually through applying transformations such as weight pruning or quantization. In this paper, we present a novel scheme for lossy weight encoding co-designed with weight simplification techniques. The encoding is based on the Bloomier filter, a probabilistic data structure that can save space at the cost of introducing random errors. Leveraging the ability of neural networks to tolerate these imperfections and by re-training around the errors, the proposed technique, named Weightless, can compress weights by up to 496× without loss of model accuracy. This results in up to a 1.51× improvement over the state-of-the-art.

1. Introduction

The continued success of neural networks comes with increasing demands on compute, memory, and networking resources. Moreover, the correlation between model size and accuracy suggests that tomorrow's networks will only grow larger. This growth presents a challenge for resource-constrained platforms such as mobile phones and wireless sensors. A common workflow is to train networks in the cloud using high-performance devices including GPUs and even customized hardware (Google, 2017). Trained neural networks are then distributed to edge devices and used to make inferences on new data. To make the inferences efficient, devices now leverage special hardware (Apple, 2017; Qualcomm, 2017), mobile GPUs, or CPUs with heavily optimized code (Hazelwood et al., 2018).

While many products have been successfully deployed to

[1]Harvard University, Cambridge, MA [2]Facebook, Menlo Park, CA. Correspondence to: Brandon Reagen <reagen@fas.harvard.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

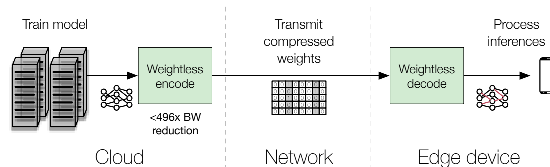

Figure 1. Weightless compresses neural networks by up to three orders of magnitude, which facilitates efficient transmission of trained models from the cloud to edge devices.

alleviate the computational costs of network training and inference, one remaining practical issue is reducing the burden of distributing the latest models, especially in regions of the world not using high-bandwidth networks. For instance, it is estimated that, globally, 800 million users will be using 2G networks by 2020 (GSMA, 2014), which can take up to 30 minutes to download just 20 MB of data. By contrast, today's neural networks are on the order of tens to hundreds of MBs, making them difficult to distribute. The limited storage capacity on resource-constrained devices poses an additional challenge as more applications look to leverage neural networks.

Model compression is a popular solution for these problems. A variety of compression algorithms have been proposed in recent years and many exploit the intrinsic redundancy in model weights. Broadly speaking, the majority of this work has focused on simplification methods (e.g., weight pruning and quantization), while comparatively little effort has been spent on devising techniques for encoding and compressing.

In this paper we propose *Weightless*: a novel lossy compression method co-designed with weight simplification techniques. Weightless is based on the probabilistic Bloomier filter data structure (Chazelle et al., 2004). Bloomier filters inexactly store a function map, and by adjusting the filter parameter, we can elect to reduce the structure's size at the cost of an increased chance of erroneous values. We use the filters to compactly encode the weights of a neural network for transmission and storage (Figure 1), exploiting redundancy in the weights to tolerate some errors. Bloomier filters excel when the encoded data is sparse and the values have a small contiguous range. Therefore, to maximize compression, we *simplify* the weights for encoding by aggressively pruning connections and clustering values. Finally, the size of the filters encoding the weights can be further reduced

using conventional compression techniques. Weightless employs arithmetic coding, an entropy optimal compression algorithm, to compress the Bloomier filters. Combined, Weightless demonstrates compression rates of up to 496$\times$ without loss of accuracy, improving state-of-the-art by up to 1.51$\times$. To conclude, we present a case study showing how Weightless' compression ratio scales better with sparsity than competing methods—as more effective pruning methods are proposed, Weightless provides even more benefit.

This work demonstrates the efficacy of compressing neural networks with *lossy* encoding using probabilistic data structures. Even after applying the aggressive lossy simplification steps of weight pruning and clustering (see Section 2), there is still sufficient extraneous information left in model weights to allow an approximate encoding scheme to substantially reduce the memory footprint without loss of model accuracy.

2. Related Work

Our goal is to minimize the static memory footprint of a neural network without compromising accuracy. Neural network weights exhibit ample redundancy, and a wide variety of techniques have been proposed to exploit this attribute. We group these techniques into two categories: (1) methods that modify the loss function or structure of a network to reduce free parameters and (2) methods that compress a given network by removing unnecessary information.

The first class of methods aim to directly train a network with a small memory footprint by introducing specialized structure or loss. Examples of specialized structure include low-rank, structured matrices of Sindhwani et al. (2015) and randomly-tied weights of Chen et al. (2015). Examples of specialized loss include teacher-student training for knowledge distillation (Bucila et al., 2006; Hinton et al., 2015) and diversity-density penalties (Wang et al., 2017). These methods can achieve significant space savings, but also typically require modification of the network structure and full retraining of the parameters.

An alternative approach, which is the focus of this work, is to compress an existing, trained model. This exploits the fact that most neural networks contain far more information than is necessary for accurate inference (Denil et al., 2013). This extraneous information can be removed to save memory. Much prior work has explored this opportunity, generally by applying a two-step process of first *simplifying* weight matrices and then *encoding* them in a more compact form.

For example, pruning by selectively zeroing weight values (LeCun et al., 1989; Guo et al., 2016) can, in some cases, eliminate over 99% of the values without penalty. Similarly, most models do not need many bits of information to represent each weight. Quantization collapses weights to a smaller set of unique values, for instance via reduction to fixed-point binary representations (Gupta et al., 2015) or clustering techniques (Gong et al., 2014).

Simplifying weight matrices can further enable the use of more compact encoding schemes, improving compression. For example, two recent works (Han et al., 2016; Choi et al., 2017) encode pruned and quantized neural networks with sparse matrix representations. In both works, however, the encoding step is a lossless transformation, applied on top of lossy simplification.

3. Weightless

Weightless is a lossy encoding scheme based around Bloomier filters. We begin by describing what a Bloomier filter is, how to construct one, and how to retrieve values from it. Next, we show how the Bloomier filter can be adapted to encode the weights of a neural network and propose a set of weight augmentations to improve compression. Finally, we recount related but ultimately unsuccessful alternative design choices we encountered while developing Weightless.

3.1. The Bloomier filter

A Bloomier filter generalizes the idea of a Bloom filter (Bloom, 1970), which are data structures that answer queries about set membership. Given a subset S of a universe U, a Bloom filter answers queries of the form, "Is $v \in S$?". If v is in S, the answer is always yes; if v is not in S, there is some probability of a false positive. By allowing false positives, Bloom filters can dramatically reduce the space needed to represent the set. A Bloomier filter (Chazelle et al., 2004) is a similar data structure but instead encodes a function. For each v in a domain S, the function has an associated value $f(v)$ in the range $R = [0, 2^r)$. Given an input v, a Bloomier filter always returns $f(v)$ when v is in S. When v is not in S, the Bloomier filter returns a null value $\perp$, except that some fraction of the time there is a false positive, and the Bloomier filter returns an incorrect, non-null value in the range R.

Decoding Let S be the subset of values in U to store, with $|S| = n$. A Bloomier filter uses a small number of hash functions (typically four), and a hash table $\mathbf{X}$ of $m = cn$ cells for some constant c (1.25 in this paper), each holding $t > r$ bits. For hash functions H_0, H_1, H_2, H_M, let $H_{0,1,2}(v) \to [0, m)$ and $H_M(v) \to [0, 2^r)$, for any $v \in U$. The table $\mathbf{X}$ is set up such that for every $v \in S$,

$$X_{H_0(v)} \oplus X_{H_1(v)} \oplus X_{H_2(v)} \oplus H_M(v) = f(v).$$

Hence, to find the value of $f(v)$, hash v four times, perform three table lookups, and exclusive-or together the four values. In practice, the four hashes of v can be reduced to

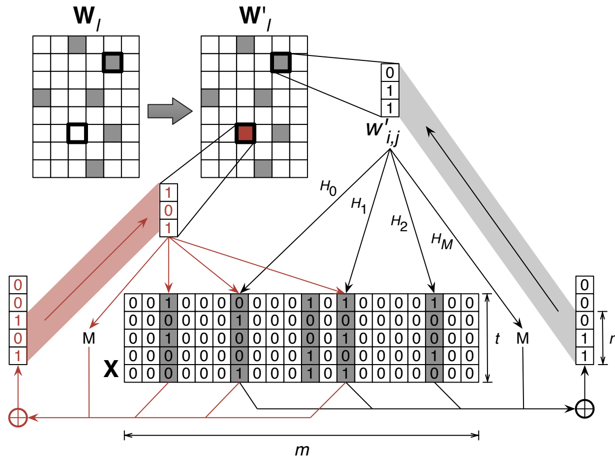

Figure 2. Encoding weights ($\mathbf{W}$) with a Bloomier filter ($\mathbf{X}$) produces a lossy reconstruction ($\mathbf{W}'$). $\mathbf{W}'$ is an inexact reconstruction of $\mathbf{W}$ from the filter $\mathbf{X}$. To retrieve the value $w'_{i,j}$, we hash its location and exclusive-or the corresponding entries of $\mathbf{X}$ together with a computed mask M. If the resulting value falls within the range $[0, 2^r)$, it is used for $w'_{i,j}$, otherwise, it is zero. The red path shows a false positive due to collisions in $\mathbf{X}$ and random M value.

a single 64-bit or 128-bit hash that is separated into four 16-bit or 32-bit values respectively. Like the Bloom filter, querying a Bloomier filter runs in $O(1)$ time. For $u \notin S$, the result, $X_{H_0(u)} \oplus X_{H_1(u)} \oplus X_{H_2(u)} \oplus H_M(u)$, will be uniform over all t-bit values. If this result is not in $[0, 2^r)$, then $\perp$ is returned; if it happened to land in $[0, 2^r)$, a false positive occurs and a result is incorrectly returned. An incorrect value is therefore returned with probability 2^{r-t}.

Encoding Constructing a Bloomier filter involves finding values for $\mathbf{X}$ such that the relationship above holds for all values in S. Known construction algorithms involve searching for suitable configurations with randomized algorithms. In their paper introducing Bloomier filters, Chazelle et al. (2004) give a greedy algorithm which takes $O(n \log n)$ time and produces a table of size $\lceil cn \rceil t$ bits with high probability. Charles & Chellapilla (2008) provide two slightly better constructions. First, they give a method with identical space requirements that runs in $O(n)$ time. They also show a separate $O(n \log n)$-time algorithm for producing a smaller table with c closer to 1. Using a more sophisticated algorithm for construction should allow for a more compact table and, by extension, improve the overall compression rate. However, we leave this for future work and use the construction method presented by Chazelle et al. (2004).

While construction (encoding) can be expensive, $O(n \log n)$), it is a one-time cost incurred after a new set of weights is trained. Moreover, the construction's absolute runtime is negligible, minutes in the case of VGG-16, compared to the time it takes to train a network. On a Intel i7-6700K CPU reconstructing (decoding) the largest layers of each model takes 0.52, 1.3, and 22.8 seconds for MNIST-300-100, LeNet5, and VGG-16 respectively; on the ARM A53 mobile class CPU used in smartphones since 2014 (Qualcomm, 2018), the same layers take 7.1, 18, and 296 seconds to reconstruct. See supplemental material for additional runtime analysis.

3.2. Approximate weight encoding with Bloomier filters

We propose using the Bloomier filter to compactly store the weights of a neural network. The function f encodes the mapping between indices of nonzero weights to their corresponding values. Given a weight matrix $\mathbf{W}$, define the domain S to be the set of indices $\{i, j \mid w_{i,j} \neq 0\}$. Likewise, the range R, corresponding to the weight values, is $[-2^{a-1}, 2^{a-1}) - \{0\}$ for a such that all values of $\mathbf{W}$ fall within the interval. Due to weight value clustering (see below) this range is remapped to $[0, 2^r)$ and encodes the cluster indices. A null response from the filter means the weight has a value of zero.

Once f is encoded in a filter, an approximation $\mathbf{W}'$ of the original weight matrix is reconstructed by querying it with all indices. The original nonzero elements of $\mathbf{W}$ are preserved in the approximation, as are *most* of the zero elements. A small subset of zero-valued weights in $\mathbf{W}'$ will take on nonzero values as a result of random collisions in $\mathbf{X}$, possibly changing the model's output. Figure 2 illustrates the operation of this scheme: an original nonzero is correctly recalled from the filter on the right and a false positive is created by an erroneous match on the left (red).

Co-designing weight simplification and Bloomier filters Because the space used by a Bloomier filter is $O(nt)$, they are especially useful under two conditions: (1) the stored function is sparse (small n, with respect to $|U|$) and (2) it has a restricted range of output values (small r, since $t > r$). To improve overall compression, we co-design weight simplification methods with these properties. In particular we leverage weight pruning and quantization.

Pruning networks to enforce sparsity (condition 1) has been studied extensively (Hassibi & Stork, 1993; LeCun et al., 1989). In this paper, we consider two different pruning techniques: (i) magnitude threshold plus retraining and (ii) dynamic network surgery (DNS) (Guo et al., 2016). Magnitude pruning with retraining is straightforward to use and offers good results. DNS is a recently proposed technique that prunes the network during training. We were able to acquire two sets of models, LeNet-300-100 and LeNet5, that were pruned using DNS and include them in our evaluation; as no reference was published for VGG-16 only magnitude pruning is used. Regardless of how it is accomplished, improving sparsity will reduce the overall encoding size linearly with the number of nonzeros with no effect on the false positive rate (which depends only on r and t). The

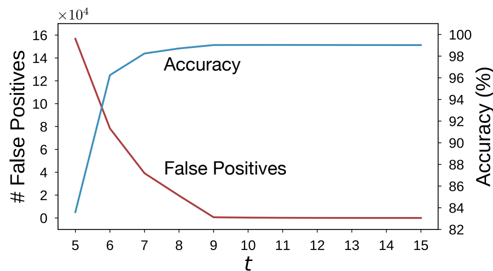

Figure 3. Trading off filter size and encoding strength. There is an exponential relationship between the t hyperparameter and both the number of false positives (red) and model accuracy (blue).

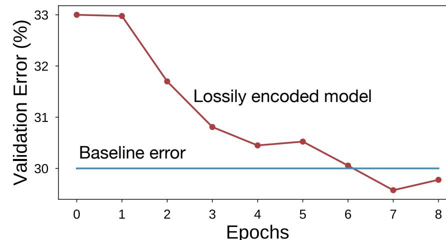

Figure 4. Retraining VGG-16 FC-1 after encoding FC-0 in a Bloomier filter. The error increases from 30% to 33% after encoding FC-0. After a few retraining epochs the error is recovered.

reason for using two methods is to demonstrate the benefits of Weightless as networks increase in sparsity, the DNS networks are notably more sparse than those simplified with magnitude pruning.

Reducing r (condition 2) amounts to restricting the range of the stored function or minimizing the number of bits required to represent weight values. Though many solutions to discretize weights exist (e.g., limited binary precision and advanced quantization techniques (Choi et al., 2017)), we use k-means clustering. After clustering the weight values, the k centroids are saved into an auxiliary table and the elements of $\mathbf{W}$ are replaced with indices into this table. This style of indirect encoding is especially well-suited to Bloomier filters, as these indices represent a small, contiguous set of integers. Another benefit of using Bloomier filters is that k does not have to be a power of 2. When reconstructing weights from Bloomier filters, the result of the XORs can be checked with an inequality, rather than a bitmask. This allows Bloomier filters to use exactly k nonzero values. In other methods, like that of Han et al. (2016), k must be a power of two (or some values are simply wasted). Bloomier filters thus allow more flexible tradeoffs between compression and the false positive rate.

Tuning the filter size The use of Bloomier filters introduces an additional hyperparameter t that sets the filters' encoding strength (i.e., the number of bits per cell in the Bloomier filter). The hyperparameter t trades off the Bloomier filter's size and the false positive rate which, in turn, affects model accuracy, see Figure 3. While t needs to be tuned, we find it far easier to reason about than other neural network hyperparameters. Because we encode k clusters, the only formal constraint is that t must be greater than $\lceil \log_2 k \rceil$. Each additional t bit reduces the number of false positives by a factor of 2, limiting the number of reasonable values for t. When t is too low, the networks experience substantial accuracy loss. However, higher values of t offer diminishing returns as weight's have some implicit resilience to errors. Experimentally, for the models considered, we find that t typically falls in the range of 6 to 9.

Retraining to mitigate the effects of false positives We encode each layer's weights sequentially. Because the weights are fixed, the Bloomier filter's false positives are deterministic. This allows for the retraining of deeper network layers to compensate for errors. It is important to note that encoded layers are not retrained (see Algorithm 1). If the encoded layer was retrained, a new encoding would have to be constructed (because changing the weights changes S) and the indices of weights that result in false positives would differ after every iteration of retraining. Instead, we find retraining all *subsequent* layers to be an effective optimization, typically allowing us to reduce t by one or two bits (e.g., going from $t = 8 \rightarrow 6$ saves 25%).

Figure 4 demonstrates the process of retraining around false positives introduced by the Bloomier filter for VGG-16's first and largest fully-connected layer (see Table 1: VGG-16, FC-0). Epoch 0 corresponds to the error of the entire network with the reconstructed approximation of the fully-connected layer's weights. Note that, as the blue line indicates the baseline accuracy, the encoding results in a 3.0% absolute increase to model validation error. As the deeper layers are retrained for a couple of epochs, the model's accuracy significantly improves, and after only 6 epochs the original and approximate weights achieve nearly indistinguishable accuracy.

Compressing Bloomier filters When sending weight matrices over a network, it is not necessary to retain the ability to access weight values as they are being sent, so it is advantageous to add another layer of compression for transmission. We use arithmetic coding, a lossless, entropy-optimal stream code which exploits the distribution of values in the table (MacKay, 2005). Because the nonzero entries in a Bloomier filter are, by design, uniformly distributed values in $[1, 2^t - 1)$, improvements from this stage largely come from the prevalence of zero entries. In this paper, we use *encoding* to refer to the use of the Bloomier filter and compression as arithmetic coding.

Algorithm 1 Weightless compression method

Result: $\mathbf{A}$, Arithmetic coded Bloomier filters
Input: M $\leftarrow$ simplified trained model with n layers
$k_i \leftarrow$ number of clusters per layer
β_{err} = evaluate(M)
for $i = 0$ **to** $n - 1$ **do**
  $t \leftarrow \lceil \log_2 k_i \rceil + 1$
  $\mathrm{W}_{P+C} \leftarrow$ K-means(M_i, k_i)
  $\omega_{err} \leftarrow \infty$
  while $\beta_{err} < \omega_{err}$ **do**
    $\mathbf{X} \leftarrow$ `BloomierEncode`(W_{P+C}, t)
    $\mathrm{M}_i \leftarrow$ `BloomierReconstruct`($\mathbf{X}$)
    `RetrainAndPrune` ($\mathrm{M}_{i+1 \to N}$)
    $\omega_{err} \leftarrow$ evaluate(M)
    $t \leftarrow t + 1$
  end while
  $\mathbf{A}_i \leftarrow$ ArithmeticCode($\mathbf{X}$)
end for
Transmit $\mathbf{A}$

3.3. The Weightless compression pipeline

The complete Weightless compression flow is given formally in Algorithm 1. Given a simplified model **M**, (i.e., a pruned and clustered neural network), a baseline error, β_{err}, is measured and saved as an accuracy target all encoding approximations must uphold. Note that if some accuracy degradation is tolerable, significant additional compression can be achieved.

Next, each layer to be compressed is encoded as a Bloomier filter starting at some minimal t value greater than the minimal number of bits needed to represent the clusters. The approximation of the weights is then computed by querying the constructed filter for all weight indexes and replaces the corresponding layer's weights in the original layer. Deeper network layers are then retrained to adapt and mitigate the effects of the errors introduced by the encoding, while simultaneously being pruned to maintain simplification criterion (1) for when they are encoded.

Once training converges, typically after a few epochs, the network error, ω_{err}, is computed and compared to the baseline β_{err}. If the Bloomier filter encoding increased error, t is increased (halving the false positives of the filter), and the process is repeated. In practice, we found setting t to be trivial given the exponential relationship between t and the number of errors. (In our experiments, t usually assumes a value between 6 and 9 for aggressively clustered weights.) Once an acceptable t is found, the layer's filter is further compressed with lossless arithmetic coding.

3.4. Ineffective optimization strategies

We experimented with techniques that ultimately did not improve compression. We briefly describe notable ones here to provide additional intuition on using approximate methods to encode neural network weights.

Encoding with Bloom filters An alternative approach to encode a weight matrix is to treat every weight $w_{i,j}$ as an r-bit binary value $\{w_{i,j,1}, \ldots, w_{i,j,r}\}$ and for each bit $w_{i,j,k}$ that equals 1, add the index (i, j, k) to a Bloom filter. We can then reconstruct an approximation $\mathbf{W}'$ of $\mathbf{W}$ by querying the filter for every bit position independently. Unlike a Bloomier filter, a Bloom filter depends on the absolute number of one-bits in a weight matrix, not the number of nonzeros. A pruned and clustered LeNet-300-100 has 15,308 one-bits, and sorting the cluster values by frequency to assign the most frequent numbers a cluster index of a power of two can save an additional 10%. We also considered retouched Bloom filters (Donnet et al., 2006). A retouched Bloom filter trades false positives for false negatives by selectively zeroing out ones in a constructed Bloom filter. In practice, the remaining nonzero weights after pruning are critical to model performance. Introducing even a few false negatives is detrimental to accuracy. As weights are exceptionally robust to false positives, the trade-off was not worthwhile. Ultimately, Bloom filters are not competitive with Bloomier encoding.

Activity pruning When erroneous weights are multiplied by a zero input they are said to be *logically masked* (Mukherjee et al., 2003). This rate can be artificially inflated by setting small-magnitude activities to zero at runtime, masking more errors. If a false positive resulted in an abnormally large erroneous weight, then even small activities might cause non-negligible errors to arise. We found this was not the case in practice. The highest magnitude weight errors were not large enough to propagate with small activities.

Exception lists In an attempt to further reduce t when using Bloomier filters, we proposed *exception lists*. Exception lists are axillary data structures built to track the indexes of select false positives. Because a Bloomier filter can only fail by having a zero weight assume a nonzero value, we only needed to store the relative Bloomier filter hit address of the false positives to fix the erroneous response and return zero instead. This proved to be space inefficient as the exception list size was on the order of the savings gained from reducing t in the first place. Minimizing t via retraining deeper layers proved to be a better alternative.

Table 1. **Experimental Setup.** Summary of error, baseline parameters (sparsity and number of clusters), and Weightless' size hyper-parameter (t) for each layer. We prune LeNet-300-100 and LeNet5 using both magnitude threshold pruning (Magnitude) and dynamic network surgery (DNS). Top-1 error is reported for all simplified models.

Model	Baseline						
	Pruning Method	Error %	Layer	Size (KB)	Nonzero %	Clusters	t
LeNet-300-100	Magnitude	1.76	FC-0	919	5.0	9	8
			FC-1	117	5.0	9	9
	DNS	2.03	FC-0	919	1.8	9	9
			FC-1	117	1.8	10	8
LeNet5	Magnitude	0.98	CNN-1	36	7.0	9	8
			FC-0	2304	5.5	9	7
	DNS	0.96	CNN-1	98	3.1	10	8
			FC-0	1564	0.73	10	8
VGG-16	Magnitude	35.9	FC-0	392000	2.99	4	6
			FC-1	64000	4.16	4	8

4. Experiments

4.1. Experimental setup

We evaluate Weightless on three networks commonly used to study compression: LeNet-300-100, LeNet5 (LeCun et al., 1998), and VGG-16 (Simonyan & Zisserman, 2015). The LeNet networks use MNIST (Lecun & Cortes, 1998) and VGG-16 uses ImageNet (Russakovsky et al., 2015). The networks are trained and tested in Keras (Chollet, 2017). The Bloomier filter was implemented in-house and uses a Mersenne Twister pseudorandom number generator for uniform hash functions. To reduce the cost of constructing the filters for VGG-16, we shard the nonzero weights into ten separate filters, which are built in parallel to reduce construction time. Sharding does not significantly affect compression or the false positive rate as long as the number of shards is small (Broder & Mitzenmacher, 2004).

Table 1 shows the models and simplification parameters used in our experiments. We apply Weightless to the largest layers in each model. This corresponds to the first two fully-connected layers of LeNet-300-100 and VGG-16. For LeNet5, the second convolutional layer and the first fully-connected layer are the largest. These layers account for 99.6%, 99.7%, and 86% of the weights in LeNet5, LeNet-300-100, and VGG-16, respectively.

In all experiments we report the compression ratio with respect to 32-bit datatypes. To compare against the state-of-the-art, we reimplemented Deep Compression (Han et al., 2016) in Keras. Deep Compression implements a lossless optimization pipeline where pruned and clustered weights are encoded using compressed sparse row encoding (CSR) and then compresses the CSR tables with Huffman coding; in this section we refer to Deep Compression as the application of CSR and Huffman coding to simplified weights. The Deep Compression baseline used here is notably better than the original publication (e.g., VGG-16 FC-0 went from $91\times$ to $119\times$).

To make the comparison commensurable, the simplified weights used as input to Weightless and Deep Compression are the same, and the resulting post-compression accuracies are also the same[1]. While Weightless does provide a trade-off between compression and model accuracy, we feel the fairest method when comparing against a lossless technique is with iso-accuracy. We provide a case study in Section 4.4 that shows the degree to which both methods can improve compression with either increased sparsity or error.

Weights are pruned using either magnitude threshold or dynamic network surgery (see Section 3.2). Once pruned, weights are clustered with k-means. We found that careful choice of initial seeds helped to minimizing the number of clusters needed. We use density-based initialization on a per-layer basis, where initial cluster values are assigned based on the input weight distribution. Han et al. (2016) reported that linear cluster initialization yielded the best results, however, we found that density based initialization helped avoid empty value ranges (i.e., regions of the weight distribution not used due to weight pruning). In the extreme, we *require only four cluster (2 bits) for VGG-16 FC-0*; this is a 60% reduction over the 5 bits used in (Han et al., 2016).

[1] Weightless' accuracy can be slightly higher due to training noise but is never worse.

Table 2. **Weight encoding.** Weights encoded using Bloomier filters (Weightless) are smaller than those encoded with CSR (Deep Compression). Bloomier filters tend to do relatively better on larger models and when using more advanced pruning algorithms (e.g., dynamic network surgery). The Improvement column shows Bloomier filters are up to 1.99× more space efficient than CSR.

Model	Pruning Method	Layer	Compression Factor (Size KB)				
			CSR		Bloomier		Improvement
LeNet-300-100	Magnitude	FC-0	40.1×	(22.9)	40.6×	(20.1)	1.01×
		FC-1	46.9×	(2.50)	56.1×	(2.09)	1.20×
	DNS	FC-0	112 ×	(8.22)	152 ×	(6.04)	1.36×
		FC-1	99.0×	(1.18)	174 ×	(0.67)	1.75×
LeNet5	Magnitude	CNN-1	40.7×	(0.89)	46.2×	(0.78)	1.14×
		FC-0	46.6×	(46.6)	66.6×	(34.6)	1.43×
	DNS	CNN-1	80.6×	(1.21)	97.8×	(1.00)	1.21×
		FC-0	224×	(6.99)	445×	(3.52)	1.99×
VGG-16	Magnitude	FC-0	81.8×	(4790)	142 ×	(2750)	1.74×
		FC-1	71.2×	(900)	74.6×	(860)	1.05×

Table 3. **Compressing encoded weights.** Weights encoded with Bloomier filters or CSR can be further compressed for transmission or storage. Below are the results of applying arithmetic coding to Bloomier filters (Weightless) and Huffman coding to CSR (Deep Compression). We find Weightless offers up to a 1.51× improvement over Deep Compression.

Model	Pruning Method	Layer	Compression Factor (Size KB)				
			Deep Compression		Weightless		Improvement
LeNet-300-100	Magnitude	FC-0	59.1×	(15.6)	60.1×	(15.3)	1.02×
		FC-1	56.0×	(2.09)	64.3×	(1.82)	1.15×
	DNS	FC-0	153×	(5.98)	174×	(5.27)	1.13×
		FC-1	129×	(0.91)	195×	(0.60)	1.51×
LeNet5	Magnitude	CNN-1	42.8×	(0.84)	51.6×	(0.70)	1.21×
		FC-0	59.1×	(39.0)	74.2×	(31.1)	1.25×
	DNS	CNN-1	89.5×	(1.09)	114.4×	(0.86)	1.28×
		FC-0	333×	(4.70)	496×	(3.16)	1.49×
VGG-16	Magnitude	FC-0	119×	(3280)	157×	(2500)	1.31×
		FC-1	88.4×	(720)	85.8×	(740)	0.97×

4.2. Encoding simplified weights

Given a simplified baseline model, we first evaluate how well Bloomier filters encode sparse weights. Results for Bloomier encoding are presented in Table 2 and show that the filters perform exceptionally well. In the extreme case, the largest fully-connected layer in LeNet5 is compressed by 445×, a 1.99× improvement over CSR.

The size of a Bloomier filter is proportional to mt, and so sparsity and clustering determine how compact they can be. Our results suggest that sparsity is more important than the number of clusters for reducing the encoding filter size. This can be seen by comparing each LeNet5 models' magnitude pruning results to those of DNS—while DNS needs additional clusters, the increased sparsity ultimately results in a substantial size reduction. We suspect this is due to the ability of neural networks to tolerate a high false positive rate. The t value used here is already on the knee of the exponential false positive curve (see Figure 3). At this point, even if k could be reduced, it is unlikely t can be since the additional encoding strength saved by reducing k does little to protect against the doubling of false positives when in this range.

A notable result from encoding is that for VGG-16 FC-0, *there are more false positives in the reconstructed weights than nonzeros in the original, simplified weights*. Using $t = 6$ results in over 6.2 million false positives while after simplification there are only 3.07 million nonzero weights. Before retraining, Bloomier filter encoding increased the top-1 error by 3.0 percentage points. This is why we see Bloomier filters work so well here–most applications cannot

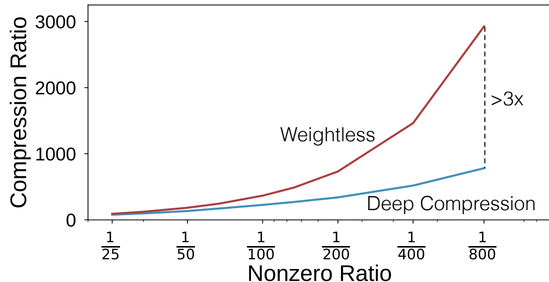

Figure 5. Weightless exploits sparsity more effectively than Deep Compression. By setting pruning thresholds in LeNet5 FC-0 to produce specific nonzero ratios, we can study how compression scales with sparsity. Weightless' accuracy guarantees from Algorithm 1 hold here.

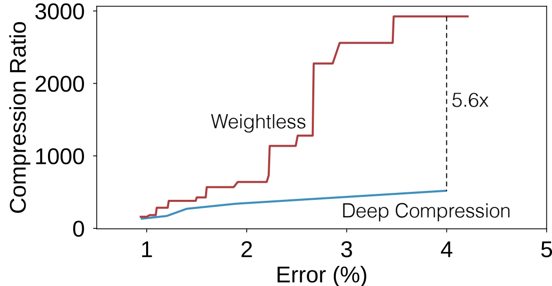

Figure 6. Weightless scales compression better with model error. This property can be leveraged in applications where trading accuracy for efficiency is advantageous. Weightless offers more compression at each iso-accuracy point. Points are generated by sweeping the pruning threshold for LeNet5 FC-0 to increase sparsity and sweeping t from 6 to 9 for Weightless.

function with this level of approximation, nor do they have an analogous retrain mechanism.

4.3. Compressing Bloomier encoded weights

When sending a model over a network, an additional stage of compression can be used to optimize for size. Deep Compression uses Huffman coding, and we propose arithmetic coding in Weightless, as described in Section 3.2. The results in Table 3 show that while Deep Compression gets relatively more benefit from a final compression stage, Weightless remains a substantially better scheme overall. A detailed analysis of the benefits from each space saving technique can be found in the supplemental material. Data includes compression without encoding as well as Bloomier with Huffman coding and CSR with arithmetic coding.

4.4. Scaling with sparsity and model error

Recent work continues to demonstrate better ways to instill sparsity in neural networks (e.g., (Guo et al., 2016; Ullrich et al., 2017; Narang et al., 2017)). Looking forward, it is useful to quantify how Weightless scales with increased sparsity. As a proxy for improved pruning techniques we set the threshold for magnitude pruning to produce varying ratios of nonzero values for LeNet5 FC-0. We then perform retraining and clustering as usual and compare the compression results of Weightless and Deep Compression. (We note that the requirement on ω_{err} in Algorithm 1 is upheld.)

Figure 5 shows that as sparsity increases, Weightless delivers far better compression ratios. Because the false positive rate of Bloomier filters is controlled independent of the number of nonzero entries and addresses are hashed, not stored, Weightless scales exceptionally well with sparsity. On the other hand, as the total number of entries in CSR decreases, the magnitude of every index pointer grows. This results in the metadata (i.e., the pointer/index tables) required to reconstruct the weights to dominate CSR's space whereas a Bloomier filter requires no structural information.

An analogous scaling argument can be made using model error. In Figure 6, we show how the two methods scale compression when accuracy is compromised using the LeNet5 model. Here, Weightless has two sources of approximation– the weights can be more aggressively pruned or t can be reduced. Deep Compression can only leverage the sparsity. The benefits from Weightless' tuning t to fit the available sparsity is immediately obvious. When considering the error range of 0.98% to 4.0%, Weightless outperforms Deep Compression by up to 5.6×.

5. Conclusion

This paper demonstrates a novel lossy encoding scheme, called Weightless, for compressing sparse weights in deep neural networks. The lossy property of Weightless stems from its use of the Bloomier filter, a probabilistic data structure for approximately encoding functions. By first simplifying a model with weight pruning and clustering, we transform its weights to best align with the properties of the Bloomier filter to maximize compression. Combined, Weightless achieves compression of up to 496×, improving the previous state-of-the-art by 1.51×.

We also see avenues for continuing this line of research. First, as better mechanisms for pruning model weights are discovered, end-to-end compression with Weightless will improve. Second, the theory community has already developed more advanced construction algorithms for Bloomier filters, which promise asymptotically better space utilization compared to the method used in this paper. Finally, by demonstrating the opportunity for using lossy encoding schemes for model compression, we hope we have opened the door for more research on encoding algorithms and novel uses of probabilistic data structures.

Acknowledgements

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. The work was also partially supported by the U.S. Government, under the DARPA CRAFT and DARPA PERFECT programs. Support was provided in part by NSF-1704834. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Michael Mitzenmacher was supported in part by NSF grants CNS-1228598, CCF-1320231, CCF-1535795, and CCF-1563710. Brandon Reagen was supported by a Siebel Scholarship. Udit Gupta was supported by the Smith family fellowship.

References

Apple. The future is here: iphone x. https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/, 2017.

Bloom, Burton H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13 (7):422–426, 1970.

Broder, Andrei and Mitzenmacher, Michael. Network applications of bloom filters: A survey. In *Internet Mathematics*, 2004.

Bucila, Cristian, Caruana, Rich, and Niculescu-Mizil, Alexandru. Model compression. In *Proceedings of KDD*, 2006.

Charles, Denis and Chellapilla, Kumar. Bloomier filters: A second look. In *European Symposium on Algorithms*, 2008.

Chazelle, Bernard, Kilian, Joe, Rubinfeld, Ronitt, and Tal, Ayellet. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004.

Chen, Wenlin, Wilson, James, Tyree, Stephen, Weinberger, Kilian Q, and Chen, Yixin. Compressing neural networks with the hashing trick. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Choi, Yoojin, El-Khamy, Mostafa, and Lee, Jungwon. Towards the limit of network quantization. In *5th International Conference on Learning Representations*, 2017.

Chollet, François. Keras. https://github.com/fchollet/keras, 2017.

Denil, Misha, Shakibi, Babak, Dinh, Laurent, Ranzato, Marc'Aurelio, and de Freitas, Nando. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems 26*, 2013.

Donnet, Benoit, Baynat, Bruno, and Friedman, Timur. Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2nd International Conference on Emerging Networking Experiments and Technologies*, 2006.

Gong, Yunchao, Liu, Liu, Yang, Ming, and Bourdev, Lubomir D. Compressing deep convolutional networks using vector quantization. *arXiv*, 1412.6115, 2014. URL http://arxiv.org/abs/1412.6115.

Google. Cloud tpu. https://cloud.google.com/tpu/, 2017.

GSMA. Half of the worlds population connected to the mobile internet by 2020, according to new gsma figures. https://www.gsma.com/newsroom/press-release/, Novemeber 2014.

Guo, Yiwen, Yao, Anbang, and Chen, Yurong. Dynamic network surgery for efficient DNNs. In *Advances in Neural Information Processing Systems 29*, 2016.

Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, and Narayanan, Pritish. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Han, Song, Mao, Huizi, and Dally, Bill. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations*, 2016.

Hassibi, Babak and Stork, David G. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 6*, 1993.

Hazelwood, Kim, Bird, Sara, Brooks, David, Chintala, Soumith, Diril, Utku, Dzhulgakov, Dmytro, Fawzy, Mohamed, Jia, BIll, Jia, Yangqing, Kalro, Aditya, Law, James, Lee, Kevin, Lu, Jason, Noordhuis, Pieter, Smelyanskiy, Misha, Xiong, Liang, and Wang, Xiaodong. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

Hinton, Geoffrey, Vinyals, Oriol, and Dean, Jeff. Distilling the knowledge in a neural network. *arXiv:1503.0253*, 2015.

Lecun, Yann and Cortes, Corinna. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

LeCun, Yann, Denker, John S., and Solla, Sara A. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*, 1989.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

MacKay, David J.C. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, fourth printing edition, 2005.

Mukherjee, Shubhendu S., Weaver, Christopher, Emer, Joel, Reinhardt, Steven K., and Austin, Todd. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

Narang, Sharan, Diamos, Greg, Sengupta, Shubho, and Elsen, Erich. Exploring sparsity in recurrent neural networks. In *5th International Conference on Learning Representations*, 2017.

Qualcomm. Snapdragon neural processing engine now available on qualcomm developer network. https://www.qualcomm.com/news/releases/2017/07/25/snapdragon-neural-processing-engine-now-available-qualcomm-developer, 2017.

Qualcomm. Snapdragon 810 processor. https://www.qualcomm.com/products/snapdragon/processors/810, 2018.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3), 2015.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations*, 2015.

Sindhwani, Vikas, Sainath, Tara, and Kumar, Sanjiv. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems 29*, 2015.

Ullrich, Karen, Meeds, Edward, and Welling, Max. Soft weight-sharing for neural network compression. In *5th International Conference on Learning Representations*, 2017.

Wang, Shengjie, Cai, Haoran, Bilmes, Jeff, and Noble, William. Training compressed fully-connected networks with a density-diversity penalty. In *5th International Conference on Learning Representations*, 2017.