Compressing Neural Networks with the Hashing Trick

Wenlin Chen[*] WENLINCHEN@WUSTL.EDU
James T. Wilson[*] J.WILSON@WUSTL.EDU
Stephen Tyree[*†] STYREE@NVIDIA.COM
Kilian Q. Weinberger[*] KILIAN@WUSTL.EDU
Yixin Chen[*] CHEN@CSE.WUSTL.EDU

[*] Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, USA
[†] NVIDIA, Santa Clara, CA, USA

Abstract

As deep nets are increasingly used in applications suited for mobile devices, a fundamental dilemma becomes apparent: the trend in deep learning is to grow models to absorb ever-increasing data set sizes; however mobile devices are designed with very little memory and cannot store such large models. We present a novel network architecture, HashedNets, that exploits inherent redundancy in neural networks to achieve drastic reductions in model sizes. HashedNets uses a low-cost hash function to randomly group connection weights into hash buckets, and all connections within the same hash bucket share a single parameter value. These parameters are tuned to adjust to the HashedNets weight sharing architecture with standard backprop during training. Our hashing procedure introduces no additional memory overhead, and we demonstrate on several benchmark data sets that HashedNets shrink the storage requirements of neural networks substantially while mostly preserving generalization performance.

1. Introduction

In the past decade deep neural networks have set new performance standards in many high-impact applications. These include object classification (Krizhevsky et al., 2012; Sermanet et al., 2013), speech recognition (Hinton et al., 2012), image caption generation (Vinyals et al., 2014; Karpathy & Fei-Fei, 2014) and domain adaptation (Glorot et al., 2011b). As data sets increase in size, so do the number of parameters in these neural networks in order to absorb the enormous amount of supervision (Coates et al., 2013). Increasingly, these networks are trained on industrial-sized clusters (Le, 2013) or high-performance graphics processing units (GPUs) (Coates et al., 2013).

Simultaneously, there has been a second trend as applications of machine learning have shifted toward mobile and embedded devices. As examples, modern smart phones are increasingly operated through speech recognition (Schuster, 2010), robots and self-driving cars perform object recognition in real time (Montemerlo et al., 2008), and medical devices collect and analyze patient data (Lee & Verma, 2013). In contrast to GPUs or computing clusters, these devices are designed for low power consumption and long battery life. Most importantly, they typically have small working memory. For example, even the top-of-the-line iPhone 6 only features a mere 1GB of RAM.[1]

The disjunction between these two trends creates a dilemma when state-of-the-art deep learning algorithms are designed for deployment on mobile devices. While it is possible to train deep nets offline on industrial-sized clusters (server-side), the sheer size of the most effective models would exceed the available memory, making it prohibitive to perform testing on-device. In speech recognition, one common cure is to transmit processed voice recordings to a computation center, where the voice recognition is performed server-side (Chun & Maniatis, 2009). This approach is problematic, as it only works when sufficient bandwidth is available and incurs artificial delays through network traffic (Kosner, 2012). One solution is to train small models for the on-device classification; however, these tend to significantly impact accuracy (Chun & Maniatis, 2009), leading to customer frustration.

This dilemma motivates *neural network compression*. Recent work by Denil et al. (2013) demonstrates that there is a surprisingly large amount of redundancy among the

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

[1] http://en.wikipedia.org/wiki/IPhone_6

weights of neural networks. The authors show that a small subset of the weights are sufficient to reconstruct the entire network. They exploit this by training low-rank decompositions of the weight matrices. Ba & Caruana (2014) show that deep neural networks can be successfully compressed into "shallow" single-layer neural networks by training the small network on the (log-) outputs of the fully trained deep network (Bucilu et al., 2006). Courbariaux et al. (2014) train neural networks with reduced bit precision, and, long predating this work, LeCun et al. (1989) investigated dropping unimportant weights in neural networks. In summary, the accumulated evidence suggests that much of the information stored within network weights may be redundant.

In this paper we propose *HashedNets*, a novel network architecture to reduce and limit the memory overhead of neural networks. Our approach is compellingly simple: we use a hash function to group network connections into hash buckets uniformly at random such that all connections grouped to the i^{th} hash bucket share the same weight value w_i. Our parameter hashing is akin to prior work in feature hashing (Weinberger et al., 2009; Shi et al., 2009; Ganchev & Dredze, 2008) and is similarly fast and requires no additional memory overhead. The backpropagation algorithm (LeCun et al., 2012) can naturally tune the hash bucket parameters and take into account the random weight sharing within the neural network architecture.

We demonstrate on several real world deep learning benchmark data sets that HashedNets can drastically reduce the model size of neural networks with little impact in prediction accuracy. Under the same memory constraint, HashedNets have more adjustable free parameters than the low-rank decomposition methods suggested by Denil et al. (2013), leading to smaller drops in descriptive power.

Similarly, we also show that for a finite set of parameters it is beneficial to "inflate" the network architecture by reusing each parameter value multiple times. Best results are achieved when networks are inflated by a factor 8–16×. The "inflation" of neural networks with HashedNets imposes no restrictions on other network architecture design choices, such as dropout regularization (Srivastava et al., 2014), activation functions (Glorot et al., 2011a; LeCun et al., 2012), or weight sparsity (Coates et al., 2011).

2. Feature Hashing

Learning under memory constraints has previously been explored in the context of large-scale learning for sparse data sets. *Feature hashing* (or the *hashing trick*) (Weinberger et al., 2009; Shi et al., 2009) is a technique to map high-dimensional text documents directly into bag-of-word (Salton & Buckley, 1988) vectors, which would otherwise require use of memory consuming dictionaries for storage of indices corresponding with specific input terms.

Formally, an input vector $\mathbf{x} \in \mathcal{R}^d$ is mapped into a feature space with a mapping function $\phi : \mathcal{R}^d \rightarrow \mathcal{R}^k$ where $k \ll d$. The mapping ϕ is based on two (approximately uniform) hash functions $h : \mathbb{N} \rightarrow \{1, \ldots, k\}$ and $\xi : \mathbb{N} \rightarrow \{-1, +1\}$ and the k^{th} dimension of the hashed input $\mathbf{x}$ is defined as $\phi_k(\mathbf{x}) = \sum_{i:h(i)=k} x_i \xi(i)$.

The hashing trick leads to large memory savings for two reasons: it can operate directly on the input term strings and avoids the use of a dictionary to translate words into vectors; and the parameter vector of a learning model lives within the much smaller dimensional $\mathcal{R}^k$ instead of $\mathcal{R}^d$. The dimensionality reduction comes at the cost of collisions, where multiple words are mapped into the same dimension. This problem is less severe for sparse data sets and can be counteracted through multiple hashing (Shi et al., 2009) or larger hash tables (Weinberger et al., 2009).

In addition to memory savings, the hashing trick has the appealing property of being sparsity preserving, fast to compute and storage-free. The most important property of the hashing trick is, arguably, its (approximate) preservation of inner product operations. The second hash function, ξ, guarantees that inner products are unbiased in expectation (Weinberger et al., 2009); that is,

$$\mathbb{E}[\phi(\mathbf{x})^\top \phi(\mathbf{x}')]_\phi = \mathbf{x}^\top \mathbf{x}'. \qquad (1)$$

Finally, Weinberger et al. (2009) also show that the hashing trick can be used to learn multiple classifiers within the same hashed space. In particular, the authors use it for multi-task learning and define multiple hash functions $\phi_1, \ldots, \phi_T$, one for each task, that map inputs for their respective tasks into one joint space. Let $\mathbf{w}_1, \ldots, \mathbf{w}_T$ denote the weight vectors of the respective learning tasks, then if $t' \neq t$ a classifier for task t' does not interfere with a hashed input for task t; *i.e.* $\mathbf{w}_t^\top \phi_{t'}(\mathbf{x}) \approx 0$.

3. Notation

Throughout this paper we type vectors in bold ($\mathbf{x}$), scalars in regular (C or b) and matrices in capital bold ($\mathbf{X}$). Specific entries in vectors or matrices are scalars and follow the corresponding convention, $i.e.$ the i^{th} dimension of vector $\mathbf{x}$ is x_i and the $(i,j)^{th}$ entry of matrix $\mathbf{V}$ is V_{ij}.

Feed Forward Neural Networks. We define the forward propagation of the ℓ^{th} layer in a neural networks as,

$$a_i^{\ell+1} = f(z_i^{\ell+1}), \quad \text{where } z_i^{\ell+1} = \sum_{j=0}^{n^\ell} V_{ij}^\ell a_j^\ell, \qquad (2)$$

where $\mathbf{V}^\ell$ is the (virtual) weight matrix in the ℓ^{th} layer. The vectors $\mathbf{z}^\ell, \mathbf{a}^\ell \in \mathcal{R}^{n^\ell}$ denote the activation units before and after transformation through the transition function $f(\cdot)$. Typical activation functions are rectifier linear unit (ReLU) (Nair & Hinton, 2010), sigmoid or tanh (LeCun et al., 2012).

4. HashedNets

In this section we present HashedNets, a novel variation of neural networks with drastically reduced model sizes (and memory demands). We first introduce our approach as a method of random weight sharing across the network connections and then describe how to facilitate it with the hashing trick to avoid any additional memory overhead.

4.1. Random weight sharing

In a standard fully-connected neural network, there are $(n^\ell\!+\!1)\!\times\! n^{\ell+1}$ weighted connections between a pair of layers, each with a corresponding free parameter in the weight matrix $\mathbf{V}^\ell$. We assume a finite memory budget per layer, $K^\ell \ll (n^\ell+1)\times n^{\ell+1}$, that cannot be exceeded. The obvious solution is to fit the neural network within budget by reducing the number of nodes $n^\ell, n^{\ell+1}$ in layers $\ell, \ell+1$ or by reducing the bit precision of the weight matrices (Courbariaux et al., 2014). However if K^ℓ is sufficiently small, both approaches significantly reduce the ability of the neural network to generalize (see Section 6). Instead, we propose an alternative: we keep the size of $\mathbf{V}^\ell$ untouched but reduce its *effective* memory footprint through *weight sharing*. We only allow exactly K^ℓ different weights to occur within $\mathbf{V}^\ell$, which we store in a weight vector $\mathbf{w}^\ell \in \mathcal{R}^{K^\ell}$. The weights within $\mathbf{w}^\ell$ are shared across multiple randomly chosen connections within $\mathbf{V}^\ell$. We refer to the resulting matrix $\mathbf{V}^\ell$ as *virtual*, as its size could be increased (*i.e.* nodes are added to hidden layer) without increasing the *actual* number of parameters of the neural network.

Figure 1 shows a neural network with one hidden layer, four input units and two output units. Connections are randomly grouped into three categories per layer and their weights are shown in the virtual weight matrices $\mathbf{V}^1$ and $\mathbf{V}^2$. Connections belonging to the same color share the same weight value, which are stored in $\mathbf{w}^1$ and $\mathbf{w}^2$, respectively. Overall, the entire network is compressed by a factor $1/4$, *i.e.* the 24 weights stored in the virtual matrices $\mathbf{V}^1$ and $\mathbf{V}^2$ are reduced to only six real values in $\mathbf{w}^1$ and $\mathbf{w}^2$. On data with four input dimensions and two output dimensions, a conventional neural network with six weights would be restricted to a single (trivial) hidden unit.

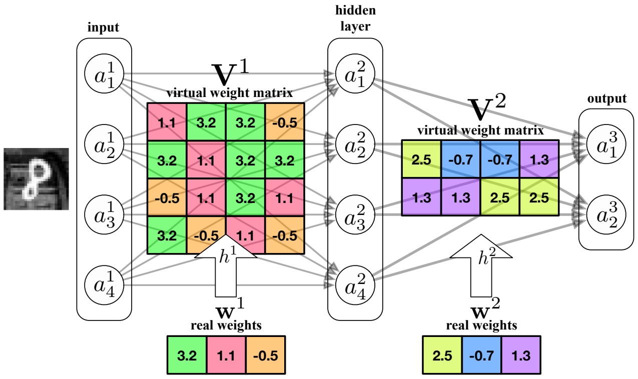

Figure 1. An illustration of a neural network with random weight sharing under compression factor $\frac{1}{4}$. The $16+9=24$ virtual weights are compressed into 6 real weights. The colors represent matrix elements that share the same weight value.

4.2. Hashed Neural Nets (HashedNets)

A naïve implementation of random weight sharing can be trivially achieved by maintaining a secondary matrix consisting of each connection's group assignment. Unfortunately, this explicit representation places an undesirable limit on potential memory savings.

We propose to implement the random weight sharing assignments using the hashing trick. In this way, the shared weight of each connection is determined by a hash function that requires no storage cost with the model. Specifically, we assign to V_{ij}^ℓ an element of $\mathbf{w}^\ell$ indexed by a hash function $h^\ell(i,j)$, as follows:

$$V_{ij}^\ell = w_{h^\ell(i,j)}^\ell, \tag{3}$$

where the (approximately uniform) hash function $h^\ell(\cdot,\cdot)$ maps a key (i,j) to a natural number within $\{1,\ldots,K^\ell\}$. In the example of Figure 1, $h^1(2,1)=1$ and therefore $V_{2,1}^1 \!=\! w^1 \!=\! 3.2$. For our experiments we use the open-source implementation *xxHash*.[2]

4.3. Feature hashing versus weight sharing

This section focuses on a single layer throughout and to simplify notation we will drop the super-scripts ℓ. We will denote the input activation as $\mathbf{a}=\mathbf{a}^\ell \in \mathcal{R}^m$ of dimensionality $m\!=\!n^\ell$. We denote the output as $\mathbf{z}=\mathbf{z}^{\ell+1} \in \mathcal{R}^n$ with dimensionality $n\!=\!n^{\ell+1}$.

To facilitate weight sharing within a feed forward neural network, we can simply substitute Eq. (3) into Eq. (2):

$$z_i = \sum_{j=1}^{m} V_{ij} a_j = \sum_{j=1}^{m} w_{h(i,j)} a_j. \tag{4}$$

Alternatively and more in line with previous work (Weinberger et al., 2009), we may interpret HashedNets in terms of feature hashing. To compute z_i, we first hash the activations from the previous layer, $\mathbf{a}$, with the hash mapping

[2] https://code.google.com/p/xxhash/

function $\phi_i(\cdot)\colon \mathcal{R}^m \rightarrow \mathcal{R}^K$. We then compute the inner product between the hashed representation $\phi_i(\mathbf{a})$ and the parameter vector $\mathbf{w}$,

$$z_i = \mathbf{w}^\top \phi_i(\mathbf{a}). \tag{5}$$

Both $\mathbf{w}$ and $\phi_i(\mathbf{a})$ are K-dimensional, where K is the number of hash buckets in this layer. The hash mapping function ϕ_i is defined as follows. The k^{th} element of $\phi_i(\mathbf{a})$, *i.e.* $[\phi_i(\mathbf{a})]_k$, is the sum of variables hashed into bucket k:

$$[\phi_i(\mathbf{a})]_k = \sum_{j:h(i,j)=k} a_j. \tag{6}$$

Starting from Eq. (5), we show that the two interpretations (Eq. (4) and (5)) are equivalent:

$$\begin{aligned} z_i &= \sum_{k=1}^{K} w_k \left[\phi_i(\mathbf{a})\right]_k = \sum_{k=1}^{K} w_k \sum_{j:h(i,j)=k} a_j \\ &= \sum_{j=1}^{m}\sum_{k=1}^{K} w_k a_j \delta_{[h(i,j)=k]} \\ &= \sum_{j=1}^{m} w_{h(i,j)} a_j. \end{aligned}$$

The final term is equivalent to Eq. (4).

Sign factor. With this equivalence between random weight sharing and feature hashing on input activations, HashedNets inherit several beneficial properties of the feature hashing. Weinberger et al. (2009) introduce an additional sign factor $\xi(i,j)$ to remove the bias of hashed inner-products due to collisions. For the same reasons we multiply (3) by the sign factor $\xi(i,j)$ for parameterizing $\mathbf{V}$ (Weinberger et al., 2009):

$$V_{ij} = w_{h(i,j)}\xi(i,j), \tag{7}$$

where $\xi(i,j)\colon \mathbb{N} \rightarrow \pm 1$ is a second hash function independent of h. Incorporating $\xi(i,j)$ to feature hashing and weight sharing does not change the equivalence between them as the proof in the previous section still holds with the sign term (details omitted for improved readability).

Sparsity. As pointed out in Shi et al. (2009) and Weinberger et al. (2009), feature hashing is most effective on sparse feature vectors since the number of hash collisions is minimized. We can encourage this effect in the hidden layers with sparsity inducing transition functions, *e.g.* rectified linear units (ReLU) (Glorot et al., 2011a) or through specialized regularization (Chen et al., 2014a; Boureau et al., 2008). In our implementation, we use ReLU transition functions throughout, as they have also been shown to often result in superior generalization performance in addition to their sparsity inducing properties (Glorot et al., 2011a).

Alternative neural network architectures. While this work focuses on general, fully connected feed forward neural networks, the technique of HashedNets could naturally be extended to other kinds of neural networks, such as recurrent neural networks (Pineda, 1987) or others (Bishop, 1995). It can also be used in conjunction with other approaches for neural network compression. All weights can be stored with low bit precision (Courbariaux et al., 2014; Gupta et al., 2015), edges could be removed (Cireşan et al., 2011) and HashedNets can be trained on the outputs of larger networks (Ba & Caruana, 2014) — yielding further reductions in memory requirements.

4.4. Training HashedNets

Training HashedNets is equivalent to training a standard neural network with equality constraints for weight sharing. Here, we show how to (a) compute the output of a hash layer during the feed-forward phase, (b) propagate gradients from the output layer back to input layer, and (c) compute the gradient over the shared weights $\mathbf{w}^\ell$ during the back propagation phase. We use dedicated hash functions between layers ℓ and $\ell+1$, and denote them as h^ℓ and ξ^ℓ.

Output. Adding the hash functions $h^\ell(\cdot,\cdot)$ and $\xi^\ell(\cdot)$ and the weight vectors $\mathbf{w}^\ell$ into the feed forward update (2) results in the following forward propagation rule:

$$a_i^{\ell+1} = f\left(\sum_{j}^{n^\ell} w^\ell_{h^\ell(i,j)}\xi^\ell(i,j)a_j^\ell\right). \tag{8}$$

Error term. Let $\mathcal{L}$ denote the loss function for training the neural network, *e.g.* cross entropy or the quadratic loss (Bishop, 1995). Further, let δ_j^ℓ denote the gradient of $\mathcal{L}$ over activation j in layer ℓ, also known as the error term. Without shared weights, the error term can be expressed as $\delta_j^\ell = \left(\sum_{i=1}^{n^{\ell+1}} V_{ij}^\ell \delta_i^{\ell+1}\right) f'(z_j^\ell)$, where $f'(\cdot)$ represents the first derivative of the transition function $f(\cdot)$. If we substitute Eq. (7) into the error term we obtain:

$$\delta_j^\ell = \left(\sum_{i=1}^{n^{\ell+1}} \xi^\ell(i,j) w^\ell_{h^\ell(i,j)} \delta_i^{\ell+1}\right) f'(z_j^\ell). \tag{9}$$

Gradient over parameters. To compute the gradient of $\mathcal{L}$ with respect to a weight w_k^ℓ we need the two gradients,

$$\frac{\partial \mathcal{L}}{\partial V_{ij}^\ell} = a_j^\ell \delta_i^{\ell+1} \text{ and } \frac{\partial V_{ij}^\ell}{\partial w_k^\ell} = \xi^\ell(i,j)\delta_{h^\ell(i,j)=k}. \tag{10}$$

Here, the first gradient is the standard gradient of a (virtual) weight with respect to an activation unit and the second gradient ties the virtual weight matrix to the actual weights through the hashed map. Combining these two, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_k^\ell} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial V_{ij}^\ell} \frac{\partial V_{ij}^\ell}{\partial w_k^\ell} \tag{11}$$

$$= \sum_{i=1}^{n^{\ell+1}} \sum_{j} a_j^\ell \delta_i^{\ell+1} \xi^\ell(i,j) \delta_{h^\ell(i,j)=k}. \tag{12}$$

5. Related Work

Deep neural networks have achieved great progress on a wide variety of real-world applications, including image classification (Krizhevsky et al., 2012; Donahue et al., 2013; Sermanet et al., 2013; Zeiler & Fergus, 2014), object detection (Girshick et al., 2014; Vinyals et al., 2014), image retrieval (Razavian et al., 2014), speech recognition (Hinton et al., 2012; Graves et al., 2013; Mohamed et al., 2011), and text representation (Mikolov et al., 2013).

There have been several previous attempts to reduce the complexity of neural networks under a variety of contexts. Arguably the most popular method is the widely used convolutional neural network (Simard et al., 2003). In the convolutional layers, the same filter is applied to every receptive field, both reducing model size and improving generalization performance. The incorporation of pooling layers (Zeiler & Fergus, 2013) can reduce the number of connections between layers in domains exhibiting locality among input features, such as images. Autoencoders (Glorot et al., 2011b) share the notion of tied weights by using the same weights for the encoder and decoder (up to transpose).

Other methods have been proposed explicitly to reduce the number of free parameters in neural networks, but not necessarily for reducing memory overhead. Nowlan & Hinton (1992) introduce soft weight sharing for regularization in which the distribution of weight values is modeled as a Gaussian mixture. The weights are clustered such that weights in the same group have similar values. Since weight values are unknown before training, weights are clustered during training. This approach is fundamentally different from HashedNets since it requires auxiliary parameters to record the group membership for every weight.

Instead of sharing weights, LeCun et al. (1989) introduce "optimal brain damage" to directly drop unimportant weights. This approach requires auxiliary parameters for storing the sparse weights and needs retraining time to fine-tune the resulting architecture. Cireşan et al. (2011) demonstrate in their experiments that randomly removing connections leads to superior empirical performance, which shares the same spirit of HashedNets.

Courbariaux et al. (2014) and Gupta et al. (2015) learn networks with reduced numerical precision for storing model parameters (*e.g.* 16-bit fixed-point representation (Gupta et al., 2015) for a compression factor of $\frac{1}{4}$ over double-precision floating point). Experiments indicate little reduction in accuracy compared with models trained with double-precision floating point representation. These methods can be readily incorporated with HashedNets, potentially yielding further reduction in model storage size.

A recent study by Denil et al. (2013) demonstrates significant redundancy in neural network parameters by directly learning a low-rank decomposition of the weight matrix within each layer. They demonstrate that networks composed of weights recovered from the learned decompositions are only slightly less accurate than networks with all weights as free parameters, indicating heavy overparametrization in full weight matrices. A follow-up work by Denton et al. (2014) uses a similar technique to speed up test-time evaluation of convolutional neural networks. The focus of this line of work is not on reducing storage and memory overhead, but evaluation speed during test time. HashedNets is complementary to this research, and the two approaches could be used in combination.

Following the line of model compression, Bucilu et al. (2006), Hinton et al. (2014) and Ba & Caruana (2014) recently introduce approaches to learn a "distilled" model, training a more compact neural network to reproduce the output of a larger network. Specifically, Hinton et al. (2014) and Ba & Caruana (2014) train a large network on the original training labels, then learn a much smaller "distilled" model on a weighted combination of the original labels and the (softened) softmax output of the larger model. The authors show that the distilled model has better generalization ability than a model trained on just the labels. In our experimental results, we show that our approach is complementary by learning HashedNets with soft targets. Rippel et al. (2014) propose a novel dropout method, nested dropout, to give an order of importance for hidden neurons. Hypothetically, less important hidden neurons could be removed after training, a method orthogonal to HashedNets.

Ganchev & Dredze (2008) are among the first to recognize the need to reduce the size of natural language processing models to accommodate mobile platform with limited memory and computing power. They propose *random feature mixing* to group features at random based on a hash function, which dramatically reduces both the number of features and the number of parameters. With the help of feature hashing (Weinberger et al., 2009), *Vowpal Wabbit*, a large-scale learning system, is able to scale to terafeature datasets (Agarwal et al., 2014).

6. Experimental Results

We conduct extensive experiments to evaluate HashedNets on eight benchmark datasets.

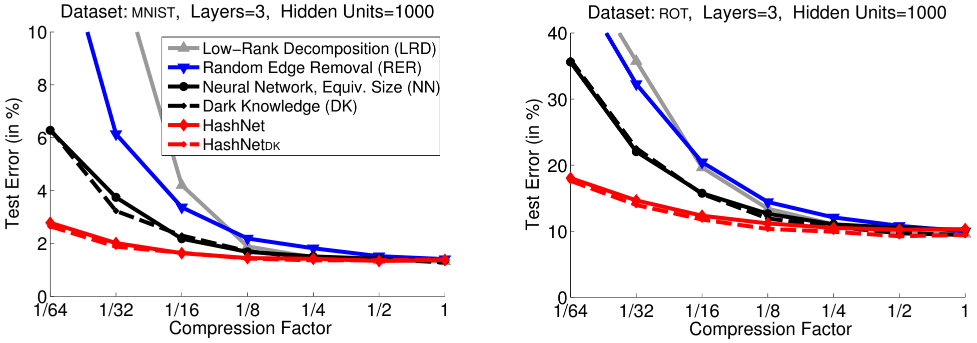

Figure 2. Test error rates under varying compression factors with 3-layer networks on MNIST (*left*) and ROT (*right*).

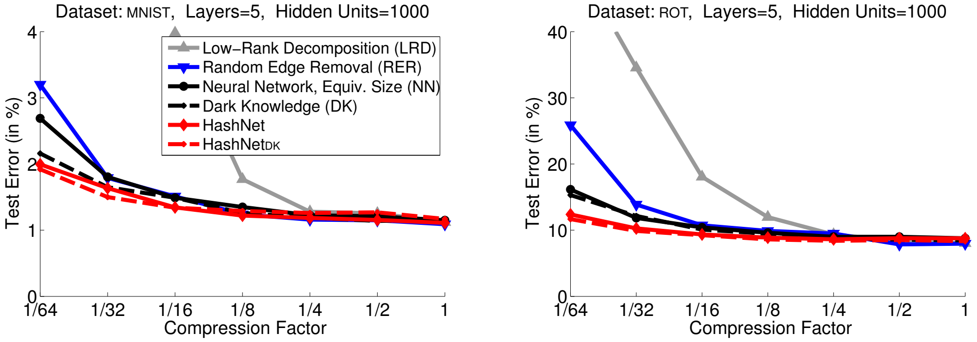

Figure 3. Test error rates under varying compression factors with 5-layer networks on MNIST (*left*) and ROT (*right*).

Datasets. Datasets consist of the original MNIST handwritten digit dataset, along with four challenging variants (Larochelle et al., 2007). Each variation amends the original through digit rotation (ROT), background superimposition (BG-RAND and BG-IMG), or a combination thereof (BG-IMG-ROT). In addition, we include two binary image classification datasets: CONVEX and RECT (Larochelle et al., 2007). All data sets have pre-specified training and testing splits. Original MNIST has splits of sizes $n = 60000$ (training) and $n = 10000$ (testing). CONVEX and RECT have 8000 and 1200 training images, respectively. And they both have 50000 testing images. Each MNIST variation set has $n = 12000$ (training) and $n = 50000$ (testing).

Baselines and method. We compare HashedNets with several existing techniques for size-constrained, feed-forward neural networks. *Random Edge Removal* (RER) (Cireşan et al., 2011) reduces the total number of model parameters by randomly removing weights prior to training. *Low-Rank Decomposition* (LRD) (Denil et al., 2013) decomposes the weight matrix into two low-rank matrices. One of these component matrices is fixed while the other is learned. Elements of the fixed matrix are generated according to a zero-mean Gaussian distribution with standard deviation $\frac{1}{\sqrt{n^\ell}}$ with n^ℓ inputs to the layer.

Each model is compared against a standard neural network with an equivalent number of stored parameters, *Neural Network (Equivalent-Size)* (NN). For example, for a network with a single hidden layer of 1000 units and a storage compression factor of $\frac{1}{10}$, we adopt a size-equivalent baseline with a single hidden layer of 100 units. For deeper networks, all hidden layers are shrunk at the same rate until the number of stored parameters equals the target size. In a similar manner, we examine *Dark Knowledge* (DK) (Hinton et al., 2014; Ba & Caruana, 2014) by training a distilled model to optimize the cross entropy with both the original labels and soft targets generated by the corresponding full neural network (compression factor 1). The distilled model structure is chosen to be same as the "equivalent-sized" network (NN) at the corresponding compression rate.

Finally, we examine our method under two settings: learning hashed weights with the original training labels (HashNet) and with combined labels and DK soft targets (HashNet$_{\text{DK}}$). In all cases, memory and storage consumption is defined strictly in terms of free parameters. As such, we count the fixed low rank matrix in the Low-Rank Decomposition method as taking no memory or storage (providing this baseline a slight advantage).

Experimental setting. HashedNets and all accompanying baselines were implemented using Torch7 (Collobert et al., 2011) and run on NVIDIA GTX TITAN graphics cards with 2688 cores and 6GB of global memory. We use 32 bit precision throughout but note that the compression

	3 Layers						5 Layers					
	RER	LRD	NN	DK	HashNet	HashNet$_{\text{DK}}$	RER	LRD	NN	DK	HashNet	HashNet$_{\text{DK}}$
MNIST	2.19	1.89	1.69	1.71	1.45	**1.43**	1.24	1.77	1.35	1.26	**1.22**	1.29
BASIC	3.29	3.73	3.19	3.18	2.91	**2.89**	2.87	3.54	2.73	2.87	**2.62**	2.85
ROT	14.42	13.41	12.65	11.93	11.17	**10.34**	9.89	11.98	9.61	9.46	8.87	**8.61**
BG-RAND	18.16	45.12	13.00	12.41	13.38	**12.27**	11.31	45.02	11.19	10.91	**10.76**	10.96
BG-IMG	24.18	38.83	20.93	19.31	22.57	**18.92**	19.81	35.06	19.33	18.94	19.07	**18.49**
BG-IMG-ROT	59.29	67.00	52.90	53.01	51.96	**50.05**	**45.67**	64.28	48.47	48.22	46.67	46.78
CONVEX	27.32	32.73	23.91	24.74	27.06	**22.93**	27.13	35.79	24.58	**23.86**	29.58	25.99
RECT	3.69	4.56	4.24	3.07	3.23	**2.96**	3.92	7.09	3.43	2.37	3.92	**2.36**

Table 1. Test error rates (in %) with a compression factor of $\frac{1}{8}$ across all data sets. Best results are printed in **blue**.

	3 Layers						5 Layers					
	RER	LRD	NN	DK	HashNet	HashNet$_{\text{DK}}$	RER	LRD	NN	DK	HashNet	HashNet$_{\text{DK}}$
MNIST	15.03	28.99	6.28	6.32	2.79	**2.65**	3.20	28.11	2.69	2.16	1.99	**1.92**
BASIC	13.95	26.95	7.67	8.44	4.17	**3.79**	5.31	27.21	4.55	4.07	3.49	**3.19**
ROT	49.20	52.18	35.60	35.94	18.04	**17.62**	25.87	52.03	16.16	15.30	12.38	**11.67**
BG-RAND	44.90	76.21	43.04	53.05	21.50	**20.32**	90.28	76.21	16.60	14.57	16.37	**13.76**
BG-IMG	44.34	71.27	32.64	41.75	26.41	**26.17**	55.76	70.85	22.77	23.59	22.22	**20.01**
BG-IMG-ROT	73.17	80.63	79.03	77.40	59.20	**58.25**	88.88	80.93	53.18	53.19	**51.93**	54.51
CONVEX	37.22	39.93	34.37	31.85	31.77	**30.43**	50.00	39.65	29.76	**26.95**	29.70	32.04
RECT	18.23	23.67	5.68	5.78	3.67	**3.37**	50.03	23.95	4.28	3.10	5.67	**2.64**

Table 2. Test error rates (in %) with a compression factor of $\frac{1}{64}$ across all data sets. Best results are printed in **blue**.

rates of all methods may be improved with lower precision (Courbariaux et al., 2014; Gupta et al., 2015). We verify all implementations by numerical gradient checking. Models are trained via stochastic gradient descent (mini-batch size of 50) with dropout and momentum. ReLU is adopted as the activation function for all models. Hyperparameters are selected for all algorithms with Bayesian optimization (Snoek et al., 2012) and hand tuning on 20% validation splits of the training sets. We use the open source Bayesian Optimization MATLAB implementation "bayesopt.m" from Gardner et al. (2014).[3]

Results with varying compression. Figures 2 and 3 show the performance of all methods on MNIST and the ROT variant with different compression factors on 3-layer (1 hidden layer) and 5-layer (3 hidden layers) neural networks, respectively. Each hidden layer contains 1000 hidden units. The x-axis in each figure denotes the fractional compression factor. For HashedNets and the low rank decomposition and random edge removal compression baselines, this means we fix the number of hidden units (n^ℓ) and vary the storage budget (K^ℓ) for the weights ($\mathbf{w}^\ell$).

We make several observations: The accuracy of HashNet and HashNet$_{\text{DK}}$ outperforms all other baseline methods, especially in the most interesting case when the compression factor is small (*i.e.* very small models). Both compression baseline algorithms, low rank decomposition and random edge removal, tend to not outperform a standard neural network with fewer hidden nodes (black line), trained with dropout. For smaller compression factors, random edge removal likely suffers due to a significant number of nodes being entirely disconnected from neighboring layers. The size-matched NN is consistently the best performing baseline, however its test error is significantly higher than that of HashNet especially at small compression rates. The use of Dark Knowledge training improves the performance of HashedNets and the standard neural network. Of all methods, only HashNet and HashNet$_{\text{DK}}$ maintain performance for small compression factors.

For completeness, we show the performance of all methods on all eight datasets in Table 1 for compression factor $\frac{1}{8}$ and Table 2 for compression factor $\frac{1}{64}$. HashNet and HashNet$_{\text{DK}}$ outperform other baselines in most cases, especially when the compression factor is very small (Table 2). With a compression factor of $\frac{1}{64}$ on average only 0.5 *bits* of information are stored per (virtual) parameter. Also note that non-neural network classifiers (Xu et al., 2015; Chen et al., 2014b) with the same model size cannot compete with this result either.

Results with fixed storage. We also experiment with the setting where the model size is fixed and the virtual network architecture is "inflated". Essentially we are fixing K^ℓ (the number of "real" weights in $\mathbf{w}^\ell$), and vary the number of hidden nodes (n^ℓ). An expansion factor of 1 denotes the case where every virtual weight has a corresponding "real" weight, $(n^\ell + 1)n^{\ell+1} = K^\ell$. Figure 4 shows the test error rate under various expansion rates of a network with one

[3] http://tinyurl.com/bayesopt

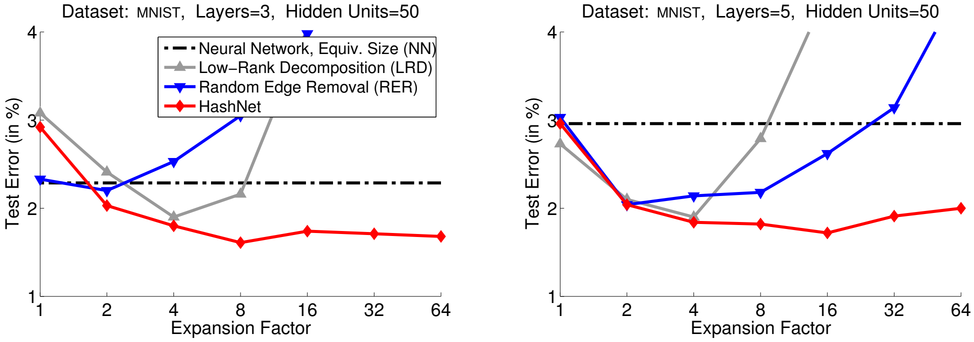

Figure 4. Test error rates with fixed storage but varying expansion factors on MNIST with 3 layers (*left*) and 5 layers (*right*).

hidden layer (*left*) and three hidden layers (*right*). In both scenarios we fix the number of real weights to the size of a standard fully-connected neural network with 50 hidden units in each hidden layer whose test error is shown by the black dashed line.

With no expansion (at expansion rate 1), different compression methods perform differently. At this point edge removal is identical to a standard neural network and matches its results. If no expansion is performed, the HashNet performance suffers from collisions at no benefit. Similarly the low-rank method still randomly projects each layer to a random feature space with same dimensionality.

For expansion rates greater 1, all methods improve over the fixed-sized neural network. There is a general trend that more expansion decreases the test error until a "sweet-spot" after which additional expansion tends to hurt. The test error of the HashNet neural network decreases substantially through the introduction of more "virtual" hidden nodes, despite that no additional parameters are added. In the case of the 5-layer neural network (right) this trend is maintained to an expansion factor of $16\times$. One could hypothetically increase n^ℓ arbitrarily for HashNet, however, in the limit, too many hash collisions would result in increasingly similar gradient updates for all weights in $\mathbf{w}$.

The benefit from expanding a network cannot continue forever. In the *random edge removal* the network will become very sparsely connected; the low-rank decomposition approach will eventually lead to a decomposition into rank-1 matrices. HashNet also respects this trend, but is much less sensitive when the expansion goes up. Best results are achieved when networks are inflated by a factor $8-16\times$.

7. Conclusion

Prior work shows that weights learned in neural networks can be highly redundant (Denil et al., 2013). HashedNets exploit this property to create neural networks with "virtual" connections that seemingly exceed the storage limits of the trained model. This can have surprising effects. Figure 4 in Section 6 shows the test error of neural networks can drop nearly 50%, from 3% to 1.61%, through expanding the number of weights "virtually" by a factor $8\times$. Although the collisions (or weight-sharing) might serve as a form of regularization, we can probably safely ignore this effect as both networks (with and without expansion) were also regularized with dropout (Srivastava et al., 2014) and the hyper-parameters were carefully fine-tuned through Bayesian optimization.

So why should additional virtual layers help? One answer is that they probably truly increase the expressiveness of the neural network. As an example, imagine we are provided with a neural network with 100 hidden nodes. The internal weight matrix has 10000 weights. If we add another set of m hidden nodes, this increases the expressiveness of the network. If we require all weights of connections to these m additional nodes to be "re-used" from the set of existing weights, it is not a strong restriction given the large number of weights in existence. In addition, the backprop algorithm can adjust the shared weights carefully to have useful values for all their occurrences.

As future work we plan to further investigate model compression for neural networks. One particular direction of interest is to optimize HashedNets for GPUs. GPUs are very fast (through parallel processing) but usually feature small on-board memory. We plan to investigate how to use HashedNets to fit larger networks onto the finite memory of GPUs. A specific challenge in this scenario is to avoid non-coalesced memory accesses due to the pseudo-random hash functions—a sensitive issue for GPU architectures.

Acknowledgements. WC and YC are supported by NSF grants CCF-1215302, IIS-1343896, DBI-1356669, CNS-1320921, and a Microsoft Research New Faculty Fellowship. KQW is supported by NSF grants IIA-1355406, IIS-1149882, EFRI-1137211. The authors thank Wenlin Wang for many helpful discussions.

References

Agarwal, Alekh, Chapelle, Olivier, Dudík, Miroslav, and Langford, John. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

Ba, Jimmy and Caruana, Rich. Do deep nets really need to be deep? In *NIPS*, pp. 2654–2662, 2014.

Bishop, Christopher M. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995.

Boureau, Y-lan, Cun, Yann L, et al. Sparse feature learning for deep belief networks. In *NIPS*, pp. 1185–1192, 2008.

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

Chen, Minmin, Weinberger, Kilian Q., Sha, Fei, and Bengio, Yoshua. Marginalized denoising auto-encoders for nonlinear representations. In *ICML*, pp. 1476–1484, 2014a.

Chen, Wenlin, Chen, Yixin, and Weinberger, Kilian Q. Fast flux discriminant for large-scale sparse nonlinear classification. In *KDD*, pp. 621–630, 2014b.

Chun, Byung-Gon and Maniatis, Petros. Augmented smartphone applications through clone cloud execution. In *HotOS*, 2009.

Cireşan, Dan C, Meier, Ueli, Masci, Jonathan, Gambardella, Luca M, and Schmidhuber, Jürgen. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011.

Coates, Adam, Ng, Andrew Y, and Lee, Honglak. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS*, 2011.

Coates, Adam, Huval, Brody, Wang, Tao, Wu, David, Catanzaro, Bryan, and Andrew, Ng. Deep learning with cots hpc systems. In *ICML*, pp. 1337–1345, 2013.

Collobert, Ronan, Kavukcuoglu, Koray, and Farabet, Clément. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Courbariaux, M., Bengio, Y., and David, J.-P. Low precision storage for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

Denil, Misha, Shakibi, Babak, Dinh, Laurent, de Freitas, Nando, et al. Predicting parameters in deep learning. In *NIPS*, 2013.

Denton, Emily, Zaremba, Wojciech, Bruna, Joan, LeCun, Yann, and Fergus, Rob. Exploiting linear structure within convolutional networks for efficient evaluation. *arXiv preprint arXiv:1404.0736*, 2014.

Donahue, Jeff, Jia, Yangqing, Vinyals, Oriol, Hoffman, Judy, Zhang, Ning, Tzeng, Eric, and Darrell, Trevor. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*, 2013.

Ganchev, Kuzman and Dredze, Mark. Small statistical models by random feature mixing. In *Workshop on Mobile NLP at ACL*, 2008.

Gardner, Jacob, Kusner, Matt, Weinberger, Kilian, Cunningham, John, et al. Bayesian optimization with inequality constraints. In *ICML*, 2014.

Girshick, Ross, Donahue, Jeff, Darrell, Trevor, and Malik, Jitendra. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua. Deep sparse rectifier networks. In *AISTATS*, 2011a.

Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *ICML*, pp. 513–520, 2011b.

Graves, Alex, Mohamed, A-R, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, and Narayanan, Pritish. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E, Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29 (6):82–97, 2012.

Hinton, Geoffrey, Vinyals, Oriol, and Dean, Jeff. Distilling the knowledge in a neural network. *NIPS workshop*, 2014.

Karpathy, Andrej and Fei-Fei, Li. Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306*, 2014.

Kosner, A.W. Client vs. server architecture: Why google voice search is also much faster than siri @ONLINE, October 2012. URL http://tinyurl.com/c2d2otr.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Larochelle, Hugo, Erhan, Dumitru, Courville, Aaron C, Bergstra, James, and Bengio, Yoshua. An empirical evaluation of deep architectures on problems with many factors of variation. In *ICML*, pp. 473–480, 2007.

Le, Quoc V. Building high-level features using large scale unsupervised learning. In *ICASSP*, pp. 8595–8598. IEEE, 2013.

LeCun, Yann, Denker, John S, Solla, Sara A, Howard, Richard E, and Jackel, Lawrence D. Optimal brain damage. In *NIPS*, 1989.

LeCun, Yann A, Bottou, Léon, Orr, Genevieve B, and Müller, Klaus-Robert. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

Lee, Kyong Ho and Verma, Naveen. A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals. *Solid-State Circuits, IEEE Journal of*, 48 (7):1625–1637, 2013.

Mikolov, Tomas, Sutskever, Ilya, Chen, Kai, Corrado, Greg S, and Dean, Jeff. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

Mohamed, Abdel-rahman, Sainath, Tara N, Dahl, George, Ramabhadran, Bhuvana, Hinton, Geoffrey E, and Picheny, Michael A. Deep belief networks using discriminative features for phone recognition. In *ICASSP*, 2011.

Montemerlo, Michael, Becker, Jan, Bhat, Suhrid, Dahlkamp, Hendrik, Dolgov, Dmitri, Ettinger, Scott, Haehnel, Dirk, Hilden, Tim, Hoffmann, Gabe, Huhnke, Burkhard, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *ICML*, pp. 807–814, 2010.

Nowlan, Steven J and Hinton, Geoffrey E. Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493, 1992.

Pineda, Fernando J. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59 (19):2229, 1987.

Razavian, Ali Sharif, Azizpour, Hossein, Sullivan, Josephine, and Carlsson, Stefan. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshop*, 2014.

Rippel, Oren, Gelbart, Michael A, and Adams, Ryan P. Learning ordered representations with nested dropout. *arXiv preprint arXiv:1402.0915*, 2014.

Salton, Gerard and Buckley, Christopher. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

Schuster, Mike. Speech recognition for mobile devices at google. In *PRICAI 2010: Trends in Artificial Intelligence*, pp. 8–10. Springer, 2010.

Sermanet, Pierre, Eigen, David, Zhang, Xiang, Mathieu, Michaël, Fergus, Rob, and LeCun, Yann. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

Shi, Qinfeng, Petterson, James, Dror, Gideon, Langford, John, Smola, Alex, and Vishwanathan, S.V.N. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, December 2009.

Simard, Patrice Y, Steinkraus, Dave, and Platt, John C. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 2, pp. 958–958. IEEE Computer Society, 2003.

Snoek, Jasper, Larochelle, Hugo, and Adams, Ryan P. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1): 1929–1958, 2014.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

Weinberger, Kilian, Dasgupta, Anirban, Langford, John, Smola, Alex, and Attenberg, Josh. Feature hashing for large scale multitask learning. In *ICML*, 2009.

Xu, Zhixiang, Gardner, Jacob R, Tyree, Stephen, and Weinberger, Kilian Q. Compressed support vector machines. *arXiv preprint arXiv:1501.06478*, 2015.

Zeiler, Matthew D and Fergus, Rob. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.

Zeiler, Matthew D and Fergus, Rob. Visualizing and understanding convolutional networks. In *ECCV*, 2014.