Training Sparse Neural Networks

Suraj Srinivas Akshayvarun Subramanya R. Venkatesh Babu
Video Analytics Lab
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore

Abstract

The emergence of Deep neural networks has seen human-level performance on large scale computer vision tasks such as image classification. However these deep networks typically contain large amount of parameters due to dense matrix multiplications and convolutions. As a result, these architectures are highly memory intensive, making them less suitable for embedded vision applications. Sparse Computations are known to be much more memory efficient. In this work, we train and build neural networks which implicitly use sparse computations. We introduce additional gate variables to perform parameter selection and show that this is equivalent to using a spike-and-slab prior. We experimentally validate our method on both small and large networks which result in highly sparse neural network models.

1. Introduction

For large-scale tasks such as image classification, large networks with many millions of parameters are often used [12], [21], [25]. However, not all of these parameters are required to achieve high performance. Recent works [6], [9] have shown that such large, deep neural networks contain lots of redundant parameters. As a result, it is possible to prune many parameters from networks without impacting performance. This would result in network architectures which make use of sparse computations. What advantages do sparse computations present? Apart from having fewer number of parameters to store ($\mathcal{O}(mn)$ to $\mathcal{O}(k)$)[1], sparse computations also decrease feedforward evaluation time ($\mathcal{O}(mnp)$ to $\mathcal{O}(kp)$)[2]. This increased efficiency of computation is crucial for applications which run on low-capacity hardware such as embedded platforms. Running heavy computations on such small devices also largely impacts processing time (making it difficult to achieve real-time performance) and increases energy requirements[9]. These lead to further challenges when designing such an embedded system[24]. While there are several strategies to exploit neural network redundancy (see Related Work section), in this work we explore the idea of pruning the weights of neural networks. This approach seems most promising given the performance of recent weight-pruning algorithms [9], when compared to other approaches. In our work we enforce weight sparsity by using suitable regularizers.

Regularizers are often used in machine learning to discourage overfitting on the data. These usually restrict the magnitude (ℓ_2/ℓ_1) of weights. However, to restrict the computational complexity of neural networks, we need a regularizer which restricts the total number of parameters of a network. A common strategy to obtain sparse parameters is to apply sparsity-inducing regularizers such as the ℓ_1 penalty on the parameter vector. However, this is often insufficient to induce sparsity for large non-convex problems like deep neural network training [5]. The contribution of this paper is to be able to induce sparsity in a tractable way for such models.

The overall contributions of the paper are as follows.

- We propose a novel regularizer that restricts the total number of parameters in the network. (Section 2)
- We perform experimental analysis to understand the behaviour of our method. (Section 4)
- We apply our method on LeNet-5, AlexNet and VGG-16 network architectures to achieve sparse neural networks. (Section 4)

2. Related Work

There have been many recent works which perform compression of neural networks. Weight-pruning techniques were popularized by LeCun *et al.*[14] and Hassibi *et al.*[10], who introduced *Optimal Brain Damage* and *Optimal Brain Surgery* respectively. Recently, Srinivas and Babu [22] proposed a neuron pruning technique, which relied on neuronal

[1]For a matrix of size $m \times n$ with k non-zero elements
[2]For matrix-vector multiplies with a dense vector of size p

2160-7516/17 $31.00 © 2017 IEEE
DOI 10.1109/CVPRW.2017.61

IEEE computer society

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.

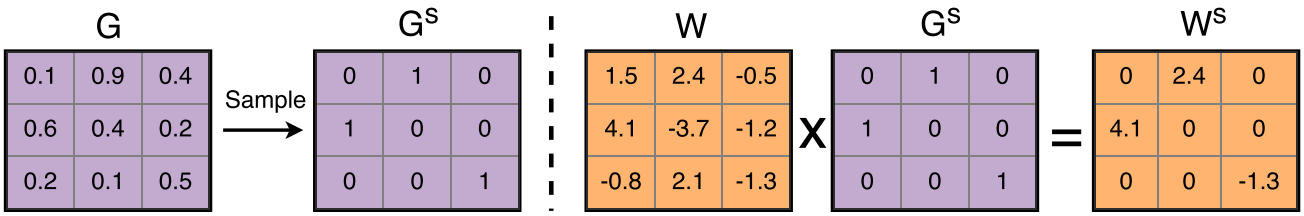


Figure 1: Our strategy for sparsifying weight matrices. First, we sample / threshold the *learnt* gate variables. We then perform element-wise multiplication of the resulting binary matrix with W, to yield a sparse matrix W^s i.e ($W^s = W \odot G^s$)

similarity. In contrast, we perform weight pruning based on learning, rather than hand-crafted rules.

Previous attempts have also been made to sparsify neural networks. Han *et al.*[9] create sparse networks by alternating between weight pruning and network training. A similar strategy is followed by Collins and Kohli [5]. On the other hand, our method performs both weight pruning and network training **simultaneously**. Further, our method has considerably less number of hyper-parameters to determine (λ_1, λ_2) compared to the other methods, which have n thresholds to be set for each of the n layers in a neural network.

Another way to perform compression is to train a smaller model to mimic a larger model. Bucilua *et al.*[3] proposed a way to achieve the same - and trained smaller models which had accuracies similar to larger networks. Ba and Caruana [1] used the approach to show that shallower (but much wider) models can be trained to perform as well as deep models. *Knowledge Distillation* (KD) by Hinton *et al.*[11] is a more general approach, of which Bucila *et al.*'s is a special case. *FitNets* by Romero *et al.*[19] use a KD-like method at several layers to learn networks which are deeper but thinner (in contrast to Ba and Caruana's shallow and wide), and achieve high levels of compression on trained models. It is possible, in principle, to perform knowledge distillation to train sparse networks that mimic deep networks. Our work "evolves" sparser networks from dense networks, and as a result can be thought of as a form of continuous knowledge transfer.

Many methods have been proposed to train models that are deep, yet have a lower parameterisation than conventional networks. Denil *et al.*[6] demonstrated that most of the parameters of a model can be *predicted* given only a few parameters. At training time, they learn only a few parameters and predict the rest. Yang *et al.*[27] propose an *Adaptive Fastfood transform*, which is an efficient re-parametrization of fully-connected layer weights. This results in a reduction of complexity for weight storage and computation. Novikov *et al.*[18] use tensor decompositions to obtain a factorization of tensors with small number of parameters. Cheng *et al.*[4] make use of circulant matrices to re-paramaterize fully connected layers. Some recent works have also focussed on using approximations of weight matrices to perform compression. Gong *et al.*[8] use a clustering-based product quantization approach to build an indexing scheme that reduces the space occupied by the matrix on disk. Note that to take full advantage of these methods, one needs to have fast implementations of the specific parameterization used. One the other hand, we use a sparse parameterization, fast implementations of which are available on almost every platform. Srinivas *et al.*[23] proposed Architecture Learning, which tried to minimize the total number of neurons in the network. However, we minimize the total number of weights in the network.

3. Problem Formulation

To understand the motivation behind our method, let us first define our notion of computational complexity of a neural network.

Let $\Phi = \{g_1^s, g_2^s, ..., g_m^s\}$ be a set of m vectors. This represents an m-layer dense neural network architecture where g_i^s is a vector of parameter indices for the i^{th} layer, i.e; $g_i^s = \{0, 1\}^{n_i}$. Here, each layer g_i^s contains n_i elements. Zero indicates absence of a parameter and one indicates presence. Thus, for a dense neural network, g_i is a vector of all ones, i.e.; $g_i^s = \{1\}^{n_i}$. For a sparse parameter vector, g_i^s would consist of mostly zeros. Let us call Φ as the index set of a neural network.

For these vectors, our notion of complexity is simply the total number of parameters in the network.

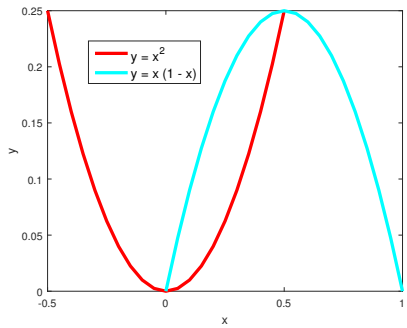
Definition 1. *The complexity of a m-layer neural network with index set Φ is given by $\|\Phi\| = \sum_{i=1}^{m} n_i$.*

We now aim to solve the following optimization problem.

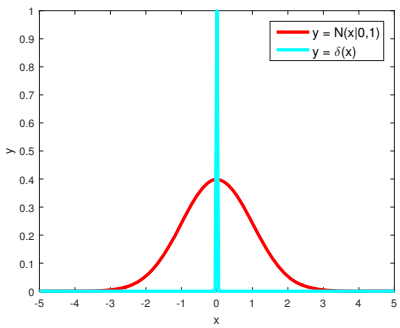
$$\hat{\theta}, \hat{\Phi} = \arg\min_{\theta, \Phi} \ell(\hat{y}(\theta, \Phi), y) + \lambda \|\Phi\| \tag{1}$$

where θ denotes the weights of the neural network, and Φ the index set. $\ell(\hat{y}(\theta, \Phi), y)$ denotes the loss function, which depends on the underlying task to be solved. Here, we learn both the weights as well as the index set of the neural network. Using the formalism of the index set, we are able

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.



(a) Bi-modal regularizer



(b) Spike and slab prior

Figure 2: **(a)** The bi-modal regularizer used in our work. Note that this encourages values to be close to 0 and 1, in contrast to a regular ℓ_2 regularizer. **(b)** An example of a spike-and-slab prior similar to the one used in this work.

to penalize the total number of network parameters. While easy to state, we note that this problem is difficult to solve, primarily because Φ contains elements $\in \{0, 1\}$.

3.1. Gate Variables

How do we incorporate the index set formalism in neural networks? Assume that the index set (G^s in Fig. 1) is multiplied pointwise with the weight matrix. This results in a weight matrix that is *effectively* sparse, if the index set has lots of zeros rather than ones. In other words, we end up learning two sets of variables to ensure that one of them - weights - becomes sparse. How do we learn such binary parameters in the first place ?

To facilitate this, we interpret index set variables (G^s) as draws from a bernoulli random variable. As a result, we end up learning the real-valued bernoulli parameters (G in Fig. 1), or *gate variables* rather than index set variables themselves. Here the sampled binary gate matrix G^s corresponds exactly to the index set, or the Φ matrix described above. To clarify our notation, G and g stand for the real-valued gate variables, while the superscript $(.)^s$ indicates binary sampled variables.

When we draw from a bernoulli distribution, we have two choices - we can either perform a *unbiased* draw (the usual sampling process), or we can perform a so-called *maximum-likelihood (ML)* draw. The ML draw involves simply thresholding the values of G at 0.5. To ensure determinism, we use the ML draw or thresholding in this work.

3.2. Promoting Sparsity

Given our formalism of gate variables, how do we ensure that the learnt bernoulli parameters are low - or in our case - mostly less than 0.5 ? One plausible option is to use the ℓ_2 or the ℓ_1 regularizer on the gate variables. However, this does not ensure that there will exist values greater than 0.5. To accommodate this, we require a *bi-modal* regularizer, i.e; a regularizer which ensures that some values are large, but most values are small. This can be achieved using a regularizer given by $w \times (1 - w)$. This was introduced by [17] to learn binary values for parameters. However, what is important for us is that this regularizer has the *bi-modal* property mentioned earlier, as shown in Figure 2a

Our overall regularizer is simply a combination of this *bi-modal* regularizer as well the traditional ℓ_2 or ℓ_1 regularizer for the individual gate variables. Our objective function is now stated as follows.

$$\hat{\theta}, \hat{\Phi} = \arg\min_{\theta,\Phi} \ell(\hat{y}(\theta, \Phi), y) + \lambda_1 \sum_{i=1}^{m} \sum_{j=1}^{n_i} g_{i,j}(1 - g_{i,j}) + \lambda_2 \sum_{i=1}^{m} \sum_{j=1}^{n_i} g_{i,j} \tag{2}$$

where $g_{i,j}$ denotes the j^{th} gate parameter in the i^{th} layer. Note that for $g_{i,j} \in \{0, 1\}$, the second term in Eqn. 2 vanishes and the third term becomes $\lambda \|\Phi\|$, thus reducing to Eqn.1.

3.3. An Alternate Interpretation

Now that we have arrived at the objective function in Eqn.2, it is natural to ask the question - how do we know that it solves the original objective in Eqn.1 ? We shall now derive Eqn.2 from this perspective.

Assuming the formulation of gate variables, we can rewrite the objective in Eqn.1 as follows.

$$\begin{aligned} \hat{\theta}, \hat{\Phi} &= \arg\min_{\theta,G} \ell(\hat{y}(\theta, G^s), y) + \lambda \sum_{i=1}^{m} \sum_{j=1}^{n_i} g^s_{i,j} \\ g^s_{i,j} &\sim \text{bernoulli}(g_{i,j}),\ \forall\, i, j \end{aligned} \tag{3}$$

where g^s is the sampled version of gate variables g. Note that Eqn.3 is a stochastic objective function, arising from

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.

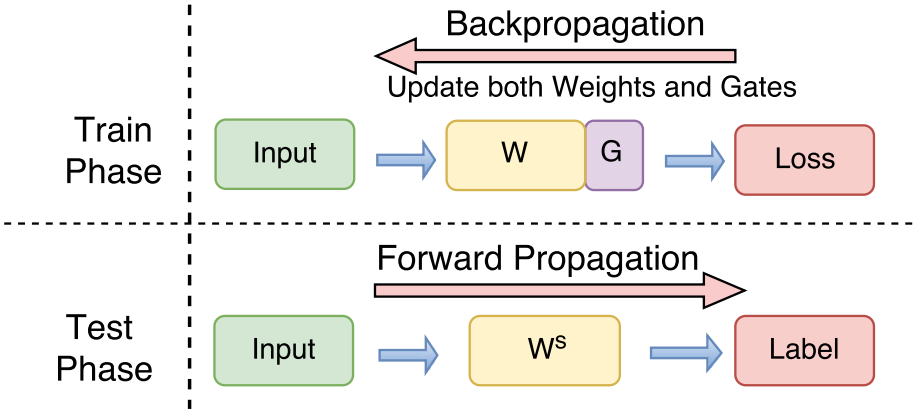


Figure 3: Our method uses different computational graphs during train and test time. During training we use both weights (W) and gate (G) variables, while during testing we directly use the resultant sparse weights ($W^s = W \odot G^s$)

the fact that g^s is a random variable. We can convert this to a real-valued objective by taking expectations. Note that expectation of the loss function is difficult to compute. As a result, we approximate it with a Monte-Carlo average.

$$\begin{aligned} \hat{\theta}, \hat{\Phi} &= \arg\min_{\theta, G} \frac{1}{t} \sum_{t} (\ell(\hat{y}(\theta, G^s), y)) + \lambda \sum_{i=1}^{m} \sum_{j=1}^{n_i} g_{i,j} \\ g^s_{i,j} &\sim \text{bernoulli}(g_{i,j}),\ \forall\, i, j \end{aligned}$$

where $\mathbf{E}(g^s_{i,j}) = g_{i,j}$. While this formulation is sufficient to solve the original problem, we impose another condition on this objective. We would like to minimize the number of Monte-Carlo evaluations in the loss term. This amounts to reducing $[\frac{1}{t}\sum_t(\ell(\hat{y}(\theta, G^s), y)) - \mathbf{E}(\ell(\hat{y}(\theta, G^s), y))]^2$ for a fixed t, or reducing the variance of the loss term. This is done by reducing the variance of g^s, the only random variable in the equation. To account for this, we add another penalty term corresponding to $\mathbf{Var}(g^s) = g \times (1 - g)$. Imposing this additional penalty and then using $t = 1$ gives us back Eqn.2.

3.4. Relation to Spike-and-Slab priors

We observe that our problem formulation closely resembles spike-and-slab type priors used in Bayesian statistics for variable selection [15]. Broadly speaking, these priors are mixtures of two distributions - one with very low variance (spike), and another with comparatively large variance (slab). By placing a large mass on the spike, we can expect to obtain parameter vectors with large sparsity.

Let us consider for a moment using the following prior for weight matrices of neural networks.

$$P(W) = \frac{1}{Z} \prod_i exp(-\ (1 - \delta(w_i)))^{\alpha}\ \mathcal{N}(w_i|0, \sigma^2)^{1-\alpha} \tag{4}$$

Here, $\delta(\cdot)$ denotes the dirac delta distribution, and Z denotes the normalizing constant, and α is the mixture coefficient. Also note that like [15], we assume that $w_i \in [-k, k]$ for some $k > 0$. This is visualized in Fig. 2b. Note that this is a multiplicative mixture of distributions, rather than additive. By taking negative logarithm of this term and ignoring constant terms, we obtain

$$-\log P(W) = -\alpha \sum_i (1 - \delta(w_i)) + \frac{1-\alpha}{2\sigma^2} \sum_i w_i^2 \tag{5}$$

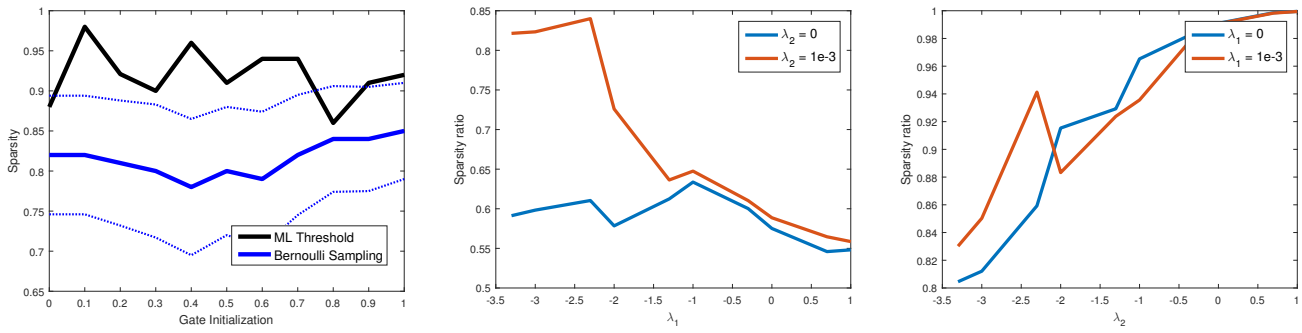
Note that the first term in this expression corresponds exactly to the number of non-zero parameters, i.e; the $\lambda \|\Phi\|$ term of Eqn. 1. The second term corresponds to the usual ℓ_2 regularizer on the weights of the network (rather than gates). As a result, we conclude that Eqn. 4 is a spike-and-slab prior which we implicitly end up using in this method.

3.5. Estimating gradients for gate variables

How do we estimate gradients for gate variables, given that they are binary stochastic variables, rather than real-valued and smooth? In other words, how do we backpropagate through the bernoulli sampling step? Bengio *et al.* [2] investigated this problem and empirically verified the efficacy of different possible solutions. They conclude that the simplest way of computing gradients - the *straight-through* estimator works best overall. Our experiments also agree with this observation.

The *straight-through* estimator simply involves back-propagating through a stochastic neuron as if it were an

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.



(a) Effect of Gate initialization (b) Effect of varying λ_1 (c) Effect of varying λ_2

Figure 4: **(a)** We vary the initialization of the gate variables and observe it's effect on sparsity. The dotted blue lines denote the variance of sparsity in case of the sampling-based method. **(b)** λ_1 seems to have a *stabilizing* effect on sparsity whereas **(c)** increasing λ_2 seems to increase sparsity. For both (b) and (c) x-axis is in $\log_{10}$ scale.

identity function. If the sampling step discussed above is given by $g^s \sim bernoulli(g)$, then the gradient $\frac{dg^s}{dg} = 1$ is used.

Another issue of consideration is that of ensuring that g always lies in $[0, 1]$ so that it is a valid bernoulli parameter. Bengio *et al.* [2] use a sigmoid activation function to achieve this. Our experiments showed that clipping functions worked better. This can be thought of as a 'linearized' sigmoid. The clipping function is given by the following expression.

$$clip(x) = \begin{cases} 1, & x \geq 1 \\ 0, & x \leq 0 \\ x, & otherwise \end{cases}$$

The overall sampling function is hence given by $g^s \sim bernoulli(clip(g))$, and the straight-through estimator is used to estimate gradients overall.

3.6. Comparison with LASSO

LASSO is a commonly used method to attain sparsity and perform variable selection. The main difference between the above method and LASSO is that LASSO is primarily a shrinkage operator, i.e.; it shrinks all parameters until lots of them are close to zero. This is not true for the case of spike-and-slab priors, which can have high sparsity and encourage large values at the same time. This is due to the richer parameterization of these priors.

3.7. Practical issues

In this section we shall discuss some practical issues pertaining to our method. Our method ironically uses twice the number of parameters as a typical neural network, as we have two sets of variables - weights and gates. As a result, model size doubles while training. However, we multiply them to result in sparse matrices which considerably reduces model size. Also we do not have to store both sets of parameters while testing, only a element-wise product of the two sets of variables is required as shown in Figure 3. Even though the model size doubles at train time, we note that speed of training / feedforward evaluation is not affected due to the fact that only element-wise operations are used.

Our method can be applied to both convolutional tensors as well as fully connected matrices. However while performing compression, we note that convolutional layers are less susceptible to compression than fully connected layers due to the small number of parameters they possess.

Layers	Initial Params	Final Params	Sparsity (%)
conv1	0.5K	0.04K	91
conv2	25K	1.78K	92.8
fc1	400K	15.4K	96.1
fc2	5K	0.6K	86.8
Total	431K	17.9K	95.84

Table 1: Compression results for LeNet-5 architecture.

4. Experiments

In this section we perform experiments to evaluate the effectiveness of our method. First, we perform some experiments designed to understand typical behaviour of the method. These experiments are done primarily on LeNet-5 [13]. Second, we use our method to perform network compression on three standard networks - LeNet-5[13],AlexNet[12] and VGG-16[21]. LeNet-5 was trained on MNIST[13] while AlexNet and VGG-16 was trained on ILSVRC-2012 dataset[20] respectively. Our implemen-

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.

Method	Parameters	Accuracy(%)	Compression Rate(%)
Baseline model	431K	99.20	1x
SVD(rank-10)[7]	43.6K	98.47	10x
Architecture Learning [23]	40.9K	99.04	10.5x
Fastfood-1024 [27]	38.8K	99.29	11x
Han *et al.*[9]	36K	99.23	12x
Result-1 of proposed method	18K	99.19	24x
Result-2 of proposed method	22K	99.33	19x

Table 2: Comparison of compression performance on LeNet-5 architecture.

tation is based on Lasagne, a Theano-based library[26].

4.1. Analysis of Proposed method

We shall now describe experiments to analyze the behaviour of our method. First, we shall analyze the effect of hyper-parameters. Second, we study the effect of varying model sizes on the resulting sparsity.

For all analysis experiments, we consider the LeNet-5 network. LeNet-5 consists of two 5×5 convolutional layers with 20 and 50 filters, and two fully connected layers with 500 and 10 (output layer) neurons. For analysis, we only study the effects sparsifying the third fully connected layer.

Layers	Initial Params	Final Params	Sparsity (%)
conv(5 layers)	2.3M	2.3M	-
fc6	38M	1.3M	96.5
fc7	17M	1M	94
fc8	4M	1.2M	70
Total	60.9M	5.9M	90

Table 3: Layer-wise compression performance on AlexNet

4.1.1 Effect of hyper-parameters

In Section 3.1 we described that we used maximum likelihood sampling (i.e.; thresholding) instead of unbiased sampling from a bernoulli. In these experiments, we shall study the relative effects of hyper-parameters on both methods. In the sampling case, sparsity is difficult to measure as different samples may lead to slightly different sparsities. As a result, we measure expected sparsity as well the it's variance.

Our methods primarily have the following hyper-parameters: λ_1, λ_2 and the initialization for each gate value. As a result, if we have a network with n layers, we have $n + 2$ hyper-parameters to determine.

First, we analyze the effects of λ_1 and λ_2. We use different combinations of initializations for both and look at it's effects on accuracy and sparsity. As shown in Table 5, both the thresholding as well as the sparsity-based methods are similarly sensitive to the regularization constants.

Layers	Initial Params	Final Params	Sparsity(%)
conv1_1 to conv4_3 (10 layers)	6.7M	6.7M	-
conv5_1	2M	2M	-
conv5_2	2M	235K	88.2
conv5_3	2M	235K	88.2
fc6	103M	102K	99.9
fc7	17M	167K	99.01
fc8	4M	409K	89.7
Total	138M	9.85M	92.85

Table 4: Layer-wise compression performance on VGG-16

λ_1	λ_2	Sparsity (%) [T]	Avg.Sparsity (%) [S]	Variance (%) [S]
0	0	54.5	53.1	16.1
1	1	98.3	93.7	3.3
1	0	62.1	57.3	5.4
0	1	99.0	92.7	4.1

Table 5: Effect of λ parameters on sparsity. [T] denotes the threshold-based method, while [S] denotes that sampling-based method.

In Section 3.3, we saw that λ_1 roughly controls the variance of the bernoulli variables while λ_2 penalizes the mean. In Table 5, we see that the mean sparsity for the pair $(\lambda_1, \lambda_2) = (0, 1)$ is high, while that for $(1, 0)$ is considerably lower. Also, we note that the variance of $(1, 1)$ is smaller than that of $(0, 1)$, confirming our hypothesis that λ_1 controls variance.

Overall, we find that both networks are almost equally sparse, and that they yield very similar accuracies. However, the thresholding-based method is deterministic, which is why we primarily use this method.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.

Method	Parameters	Top-1 Accuracy(%)	Compression Rate
Baseline model	60.9M	57.2	1x
Neuron Pruning [22]	39.6M	55.60	1.5x
SVD-quarter-F [27]	25.6M	56.18	2.3x
Adaptive FastFood 32 [27]	22.5M	57.39	2.7x
Adaptive FastFood 16 [27]	16.4M	57.1	3.7x
ACDC [16]	11.9M	56.73	5x
Collins & Kohli [5]	8.5M	55.60	7x
Han *et al.*[9]	6.7M	57.2	9x
Proposed Method	**5.9M**	**56.96**	**10.3x**

Table 6: Comparison of compression performance on AlexNet architecture

Method	Parameters	Top-1 Accuracy(%)	Compression Rate
Baseline model	138M	68.97	1x
Han *et al.*[9]	10.3M	68.66	13x
Proposed Method	**9.85M**	**69.04**	**14x**

Table 7: Comparison of compression performance on VGG-16 architecture

To further analyze effects of λ_1 and λ_2, we plot sparsity values attained by our method by fixing one parameter and varying another. In Figure 4b we see that λ_1, or the variance-controlling hyper-parameter, mainly *stabilizes* the training by reducing the sparsity levels. In Figure 4c we see that increasing λ_2 increases the sparsity level as expected.

We now study the effects of using different initializations for the gate parameters. We initialize all gate parameters of a layer with the same constant value. We also tried stochastic initialization for these gate parameters (Eg. from a Gaussian distribution), but we found no particular advantage in doing so. As shown in Figure 4a, both methods seem robust to varying initializations, with the thresholding method consistently giving higher sparsities. This robustness to initialization is advantageous to our method, as we no longer need to worry about finding good initial values for them.

4.2. Compression Performance

We test compression performance on three different network architectures - LeNet-5, AlexNet and VGG-16.

For LeNet-5, we simply sparsify each layer. As shown in Table 1, we are able to remove about 96% of LeNet's parameters and only suffer a negligible loss in accuracy. Table 2 shows that we obtain state-of-the-art results on LeNet-5 compression. For Result-1, we used $(\lambda_1, \lambda_2) = (0.001, 0.05)$, while for Result-2, we used $(\lambda_1, \lambda_2) = (0.01, 0.1)$. These choices were made using a validation set.

Note that our method converts a dense matrix to a sparse matrix, so the total number of parameters that need to be stored on disk includes the indices of the parameters. However, for ASIC implementations, one need not store indices as they can be built into the circuit structure.

For AlexNet and VGG-16, instead of training from scratch, we fine-tune the network from pre-trained weights. For such pre-trained weights, we found it be useful to pre-initialize the gate variables so that we do not lose accuracy while fine-tuning begins. Specifically, we ensure that the gate variables corresponding to the top-$k\%$ weights in the W matrix are one, while the rest are zeros. We use this pre-initialization instead of the constant initialization described previously.

To help pruning performance, we pre-initialize fully connected gates with very large sparsity (95%) and convolutional layers with very little sparsity. This means that 95% of g^s parameters are zero, and rest are one. For $g^s = 1$, the underlying gate values were $g = 1$ and for $g^s = 0$, we used $g = 0.49$. This is to ensure good accuracy by preserving important weights while having large sparsity ratios. The resulting network ended up with a negligible amount of sparsity for convolutional layers and high sparsity for fully connected layers. For VGG-16, we pre-initialize the final two convolutional layers as well with similarly large (88%) sparsity.

We run fine-tuning on AlexNet for 30k iterations ($\sim$ 18 hours), and VGG-16 for 40k ($\sim$ 24 hours) iterations before stopping training based on the combination of compression ratio and validation accuracy. This is in contrast with [9], who take about 173 hours to fine-tune AlexNet. The original AlexNet took 75 hours to train. All wall clock numbers are reported by training on a NVIDIA Titan X GPU. As shown in Table 6 and Table 7, we obtain favourable results when compared to the other network compression / sparsification methods.

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.

5. Conclusion

We have introduced a novel method to learn neural networks with sparse connections. This can be interpreted as learning weights and performing pruning simultaneously. By introducing a learning-based approach to pruning weights, we are able to obtain the optimal level of sparsity. This enables us to compress deep neural networks and achieve sparse neural network models.

6. Acknowledgement

This work was partly supported by Robert Bosch Centre for Cyber Physical Systems (RBCCPS) Research grant, Indian Institute of Science, Bangalore.

References

[1] J. Ba and R. Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662, 2014. 2

[2] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 4, 5

[3] C. Bucilua, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006. 2

[4] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2857–2865, 2015. 2

[5] M. D. Collins and P. Kohli. Memory bounded deep convolutional networks. *CoRR*, abs/1412.1442, 2014. 1, 2, 7

[6] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013. 1, 2

[7] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014. 6

[8] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 2

[9] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015. 1, 2, 6, 7

[10] B. Hassibi, D. G. Stork, et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in Neural Information Processing Systems*, pages 164–164, 1993. 1

[11] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NIPS 2014 Deep Learning Workshop*, 2014. 2

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 1, 5

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 5

[14] Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel. Optimal brain damage. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, 1989. 1

[15] T. J. Mitchell and J. J. Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83(404):1023–1032, 1988. 4

[16] M. Moczulski, M. Denil, J. Appleyard, and N. de Freitas. Acdc: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*, 2015. 7

[17] W. Murray and K.-M. Ng. An algorithm for nonlinear optimization problems with binary variables. *Computational Optimization and Applications*, 47(2):257–288, 2010. 3

[18] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015. 2

[19] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014. 2

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 5

[21] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015. 1, 5

[22] S. Srinivas and R. V. Babu. Data-free parameter pruning for deep neural networks. In M. W. J. Xianghua Xie and G. K. L. Tam, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 31.1–31.12. BMVA Press, September 2015. 1, 7

[23] S. Srinivas and R. V. Babu. Learning neural network architectures using backpropagation. In *Proceedings of the British Machine Vision Conference (BMVC)*. BMVA Press, September 2016. 2, 6

[24] F. Stein. The challenge of putting vision algorithms into a car. In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 89–94, June 2012. 1

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 1

[26] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. 6

[27] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. *arXiv preprint arXiv:1412.7149*, 2014. 2, 6, 7

Authorized licensed use limited to: J.R.D. Tata Memorial Library Indian Institute of Science Bengaluru. Downloaded on June 07,2022 at 10:22:52 UTC from IEEE Xplore. Restrictions apply.