Pruning Networks During Training via Auxiliary Parameters

Anonymous Authors[1]

Abstract

Neural networks have perennially been limited by the physical constraints of implementation on real hardware, and the desire for improved accuracy often drives the model size to the breaking point. The task of reducing the size of a neural network, whether to meet memory constraints, inference-time speed, or generalization capabilities, is therefore well-studied. In this work, we present an extremely simple scheme to reduce model size during training, by introducing auxiliary parameters to the inputs of each layer of the neural network, and a regularization penalty that encourages the network to eliminate unnecessary variables from the computation graph. Though related to many prior works, this scheme offers several advantages: it is extremely simple to implement; the network eliminates unnecessary variables as part of training, without requiring any back-and-forth between training and pruning; and it dramatically reduces the number of parameters in the networks while maintaining high accuracy.

1. Introduction

Deep learning frameworks have transformed machine learning in many ways, and in general the state-of-the-art model for any given task is a large, overparameterized neural network. These are costly to train and deploy, and practical considerations (hardware constraints, time to compute in inference) are often pushed to the absolute limit in the service of higher prediction accuracy. The art of getting deep learning models to work also contains a certain amount of "black magic" (**?**): training a high-quality network means using a whole pile of varied explicit and implicit regularizers to help the model generalize. But in the class of regularizers, L_0 regularization (constraining the number of parameters) has a special place: it is well-motivated theoretically, but difficult to achieve in practice. Empirically, models with more parameters often generalize better, even if it's just because they are more likely to contain a sub-network that can learn to perform the task well (Frankle & Carbin, 2018).

The problem of pruning unnecessary parameters from neural networks is also well-studied (Mozer & Smolensky, 1989; LeCun et al., 1990; Karnin, 1990; Reed, 1993; Han et al., 2015a;b; Wen et al., 2016; Neklyudov et al., 2017; Louizos et al., 2017a; Frankle & Carbin, 2018; Ramanujan et al., 2020; Malach et al., 2020; Roy et al., 2020); even before models exploded in size in the last decade, there were strong reasons to eliminate unnecessary variables from a trained model. Various schemes for dropping weights have been proposed, tested, and put into practice. In this paper, we develop a technique with several attractive features. First, we focus on dropping entire neurons from the network. Dropping a single neuron in an intermediate layer allows us to remove an entire column from the weights matrix in the previous layer, and an entire row in the layer following it. This means we're multiplying smaller matrices, performing a smaller number of FLOPs, and that we don't have to develop special handling for sparse matrices. Second, we focus on pruning neurons simultaneously with training the network weights; since we prune on the fly, we don't have to train the large, unpruned network to convergence, and we avoid needing to re-train from scratch. Third, the scheme we employ is far simpler than the closet-related prior works (Louizos et al., 2017b; Srinivas et al., 2017; Xiao & Wang, 2019; Neill et al., 2022), while still maintaining strong empirical performance.

The pruning method we develop here adds auxiliary parameters which function as gates, which multiply the inputs. We add only one auxiliary parameter input to linear matrices, or one to each channel of a convolutional layer, so the number of auxiliary parameters is much less than the number of parameters in the network. As the gate variables approach zero, the network activity for the corresponding input is smoothly compressed, and when the gate variables reach exactly zero the input is entirely eliminated from the network. This allows us to prune the corresponding neuron entirely. Once we have trained to convergence, or a desired level of sparsity is achieved, these auxiliary parameters can be rolled into the weight matrices; in this way, we can look at our setup as essentially reparameterizing the standard weight

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

matrices. This reparameterization allows us to effectively achieve L_0 regularization on the neurons, and converting back to a standard parameterization for inference allows a computational speedup.

Through experiments on MNIST and CIFAR-10 we show that the number of parameters can be dramatically reduced, and the cost in model accuracy is minimal compared to the relative reduction in model complexity. We also train a density estimation model of US Census data (Dua & Graff, 2017) using RealNVP (Dinh et al., 2016). In these experiments we find that deeper flows can better model the data, which is expected, but also that our neuron-pruning technique is so successful at removing unnecessary parameters from the intermediate flow steps that a model with 64 flow steps permits a smaller number of parameters than the model with only 4 flow steps, and still fits unseen data more successfully.

The main contributions of this work are as follows.

- We present a simple, single-pass L_0 regularization technique using auxiliary parameters that function as continuous-valued gates on the neurons, and allows for hard pruning of unnecessary neurons during training.
- We demonstrate that this structured sparsity empirically reduces the number of network parameters dramatically during training, while still producing high-quality models.

2. Approach

Consider a canonical neural network, composed of L layers of artificial neurons. Each neuron implements a function of its input weights comprised of a linear transformation followed by a nonlinear activation function. Together, the neurons in layer $l \in 1, \ldots, L$ transform the outputs of the previous layer using

$$\mathbf{a_l} = \sigma(\mathbf{W a_{l-1}} + \mathbf{b}) \tag{1}$$

where $\mathbf{a_l}$ are the activations of layer l, σ is a nonlinear function (such as ReLU or sigmoid), W is a weight matrix, and b is a bias vector. If layer l has n inputs and m neurons, $W \in R^{nxm}$ and $b \in R^m$. In this notation, $\mathbf{a_0}$ are the network inputs. This network is trained using (stochastic) gradient descent to minimize a loss function, $L(X, Y|\Theta) = \frac{1}{N}\sum_i L(x_i, y_i|\Theta)$ across a data set $X = \{x_1, \ldots, x_N\}$, $Y = \{y_1, \ldots, y_N\}$, and network parameters Θ; in 1, $\Theta = \{W_1, \ldots, W_L, b_1, \ldots, b_L\}$.

To reduce the number of network parameters in a manner that allows efficient computation, we would like to introduce a penalty to our model that penalizes the overall number of inputs (roughly, the number of neurons) used across all layers,

$$L(X, Y|\Theta) = \sum_{i \in 1,\ldots,N} L(x_i, y_i|\Theta) + \lambda \sum_{l \in 0,\ldots,L-1} ||\mathbf{a_l}||_\mathbf{0}, \tag{2}$$

where $||\cdot||_0$ is the L_0 norm.

We prefer to regularize the number of neurons, instead of applying an L_0 regularization term to the weight matrices, for two reasons. First, dropping a single neuron has a dramatic impact on the number of total network parameters, because it removes entire rows and columns from the weight matrices. However, the regularization term in 2 is not differentiable, and so is not straightforward to train using stochastic gradient descent. We employ the typical workaround to this by employing auxiliary parameters, s, to derive gating magnitudes, $g = \min(1, \max(0, s))$. In layer l, we multiply the inputs by the values of our gates before passing to the weight matrices,

$$\mathbf{a_l} = \sigma(\mathbf{W}(\mathbf{a_{l-1}} \odot \mathbf{g_l}) + \mathbf{b}), \tag{3}$$

with $\odot$ being the Hadamard product. We then introduce a regularization term to our loss function which penalizes the total magnitude of auxiliary parameters in the network, but is differentiable.

$$L(X, Y|\Theta) = \sum_{i \in 1,\ldots,N} L(x_i, y_i|\Theta) + \lambda \sum_{l \in 1,\ldots,L} ||s_l||_1, \tag{4}$$

where $||\cdot||_1$ is the L_1 norm.

Since gate variables g may be exactly equal to zero, we end up pruning neurons away entirely. But as these gate variables approach zero, they act more like soft gates on the information flow through the network. This is different from prior works that use the auxiliary parameters to turn binary gates on or off (**?**Srinivas et al., 2017; Ramanujan et al., 2020), because the entire network is amenable to exact gradient descent.

2.1. Implementation

During training, we constrain s to lie in a feasible range close to the allowable range of g, $s \in [-\epsilon, 1 + \epsilon]$. Since we pass s through a hard gate, rather than a sigmoid, we are able to completely eliminate inputs from the layer. The auxiliary parameters are trained along with the weight matrices using minibatch stochastic gradient descent and the Adam optimizer.

In order to speed training, we occasionally discard the neurons that have been set to zero, re-instantiating a new network with smaller weight matrices. This provides the advantage that later epochs are running a smaller and faster network, but causes some issues preserving the optimizer state. In our experiments, we first tried simply initializing a

new optimizer, discarding the stored state, and found that doing so every K epochs did not derail training. Some work suggests that

2.2. Comparison to Prior Work

In general, we follow the approach of (Louizos et al., 2017b) to L_0 regularization, but we introduce several variations that simplify the computational complexity and speed the sparsification of network weights. In their implementation, the gates are employed stochastically - at each forward pass, a gate value in the range [0, 1] is chosen from a distribution parameterized by the auxiliary parameter s, chosen so that the probability that an input is completely eliminated ($g = 0$) increases as s decreases. Driving down s reduces the expected number of nonzero neurons in the network. The authors employ the so-called "hard concrete" distribution, which is a stretched concrete distribution truncated to lie in the range [0, 1] to tie the auxiliary parameters s to the gate parameters g, and the regularization term is the probability that $g > 0$ given the value of s. The authors note the hard concrete is only one of many possible choices for stochastic functions, and we first implemented a simpler stochastic alternative: $g = max(1, min(0, s - u))$, where $u \sim \text{Uniform}(0, 1)$. Under this simplified sampling process, the probability that a gate is turned on is exactly equal to the value of the auxiliary parameter, s, and the regularization term is exactly the same as in our proposed deterministic regularizer, Equation 4. The simplified stochastic sampling dramatically sped up the process of eliminating weights from the trained networks, and we questioned the need for the computational overhead of stochastic gates. When we switch to a deterministic gate value, the regularization term in Equation 4 can no longer be interpreted as regularizing the expected L_0 norm of the inputs across the network, but empirically it performs quite similarly.

The use of auxiliary parameters to eliminate unnecessary neurons actually has quite a long history, and similar schemes have been used in prior work (Mozer & Smolensky, 1989). Many of these works treat pruning and training as separate, alternating tasks, but the drawback to this is that we must train the full network to convergence, and then train the pruned networks; since we prune during training, the longer we train, the faster each epoch runs. This is likely to save wall-clock time compared to iterative methods, even though the auxiliary parameters add some extra computation to the network. Other related approaches use auxiliary gate variables on the weight matrices, finding sparse sub-networks that can perform the task (Frankle & Carbin, 2018; Ramanujan et al., 2020; Malach et al., 2020; Fu et al., 2021); focusing on individual parameters increases the number of auxiliary variables and may not lead to any performance advantages, as modern hardware is optimized for dense matrix multiplication. A considerable number of the former works apply discontinuous gates based on the auxiliary scores, whether selecting the top-k weights or applying a hard threshold; if these procedures are applied during training, the gradients for the gate parameters are computed using the straight-through estimator. By replacing this procedure with continuous gates, our scheme allows exact gradient descent on both the weights and auxiliary variables, and still empirically produce highly compressed networks.

3. Empirical Evaluation

3.1. Image Data

In our first experiments, we look at canonical but small image datasets: MNIST and CIFAR-10. Following the prior work, we implement L_0 regularization on the number of inputs for dense layers, and on the number of output channels for 2-D convolutional layers, and we employ standard architectures: LeNet (LeCun et al., 1998) for the MNIST dataset, and a Wide Resnet (Zagoruyko & Komodakis, 2016) for the CIFAR-10 dataset. In these experiments, we test four architectures: the baseline model is unpruned, trained with the regularization weight on the auxiliary variables set to $\lambda = 0$ (**Unpruned**). In the networks that approximate L_0 regularization during training, we compare the stochastic gating proposed by (Louizos et al., 2017b) (**Stochastic - Concrete**). We also test the simplified stochastic gate with noise drawn from a uniform distribution as described in Section 2.2 (**Stochastic - Uniform**), and finally our proposed scheme that optimizes the auxiliary variables directly, with no stochasticity (**Direct Gating**).

We find that the simpler schemes drive the number of parameters down more quickly than the prior work, and that all L_0 regularized networks perform competitively with the full, unpruned versions of the networks.

3.2. Generative Modeling with RealNVP

In our second experiments, we consider the task of generative modeling using RealNVP (Dinh et al., 2016), on the 1990 US Census dataset (Dua & Graff, 2017; Meek et al., 2001). This dataset contains 2.5M records, one row per household, with a total of 67 categorical or binary variables. We train a generative model for this data using the RealNVP framework, and test accuracy using negative log likelihood on a held-out portion of the data. In particular, we train a model on $\sim$1.8M records, and evaluate on another 100K records.

RealNVP estimates the density of a dataset using a series of nonlinear, but invertible, normalizing flows. Each step in the flow computes a continuous, invertible, non-volume-preserving (hence NVP) deformation of the variables, and eventually transforms an empirical data distribution to look

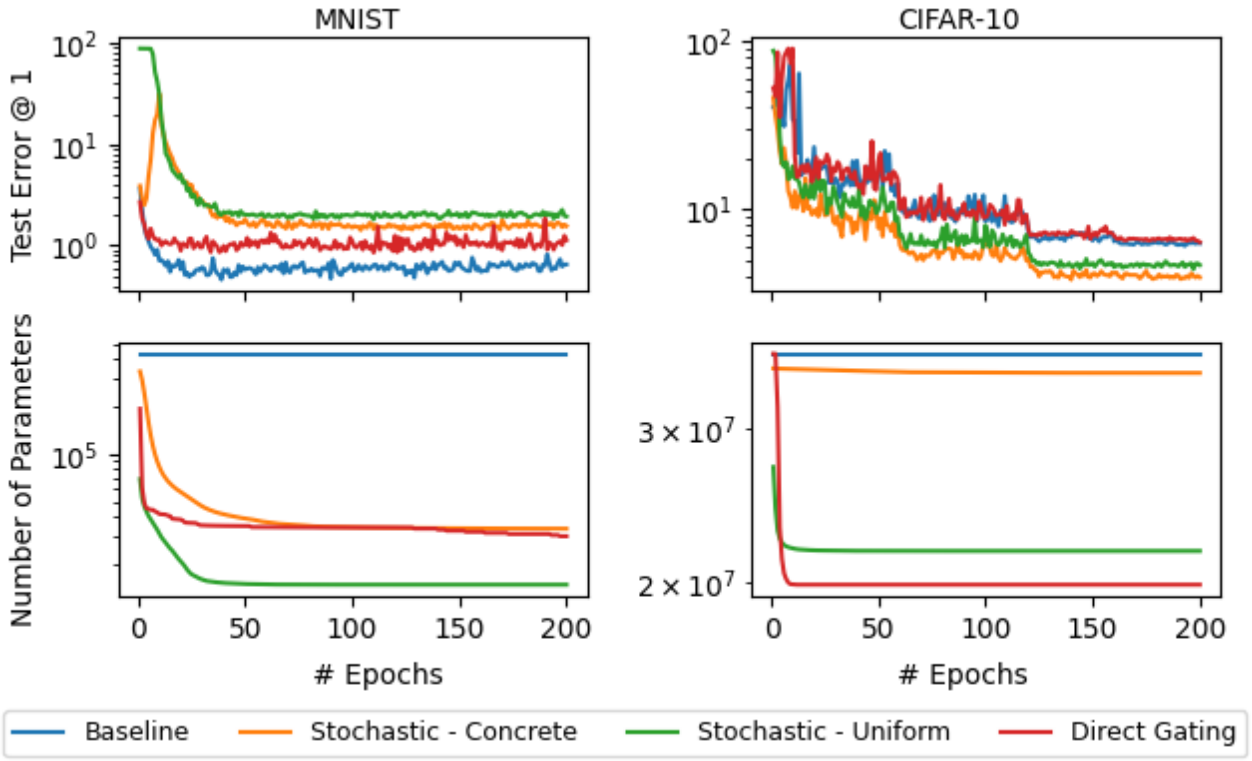

Figure 1. Model accuracy improves and size decreases during training. Our proposed scheme of directly regularizing auxiliary gate variables is competitive with, or better than, prior work on pruning neurons.

like a simple distribution - typically a spherical Gaussian of the same dimension. Since each step is invertible, the model can be used to generate new data, by running the inverse transforms, in reverse order, on a sample from a spherical Gaussian. The loss function minimized is exactly the negative log-likelihood of the data under the generative model. This is possible because the model is constructed to make each transform have a simple-to-compute determinant, which means the likelihood can be computed using the change-of-variables formula. Although the particulars of the construction and proof of its effectiveness are interesting, we'll skip the particulars and focus instead on why it is a compelling use case for automatic pruning.

Each of RealNVP's flow steps affects only a subset of the variables, using affine transformations. The scale and translation of the affine transforms are parameterized by neural networks, whose inputs are the complement of the variables being transformed. In our implementation, each flow step transforms exactly half the variables, so it takes a network depth of at least 2 to transform all the variables from a simple spherical Gaussian to anything more complex. Each each flow step contains two neural networks[1]. Logically and empirically, RealNVP tends to be better-able to match the empirical distribution with more flow steps. However, it's unclear from first principles how many parameters are required in the scale and translation networks at each step.

In Table 1 we show how our proposed L_0 regularization with auxiliary variables affects the final number of parameters and negative log likelihood compared to unpruned networks with the same initial architecture. In our experiments, we resize the networks at the end of each epoch through the dataset. We consistently find that the NLL is lower (better) for architectures with more flow steps, but that the final number of parameters does not grow linearly with flow steps, and in fact often decreases - essentially, the L_0 regularization with auxiliary variables is able to compress the individual scale and translation networks more efficiently when provided a large enough number of flow steps. Even though the unpruned network often achieves a lower overall NLL, the pruned networks have improved efficiency in representing unseen data in terms of both Bayesian Information Criterion (Neath & Cavanaugh, 2012) and Akaike Information Criterion (Sakamoto et al., 1986).

3.3. Hyperparameter Settings and Observations

In our experiments we tested an array of λ values, but found that the system performance is similar across a reasonably wide range. In the experiments with reported results, we simply use $\lambda = 1$ across all models. Raising this parameter increases the speed at which neurons are pruned in the initial

[1]Our networks use the canonical multi-layer neural network described in Section 2 with hyperbolic tangent nonlinearities

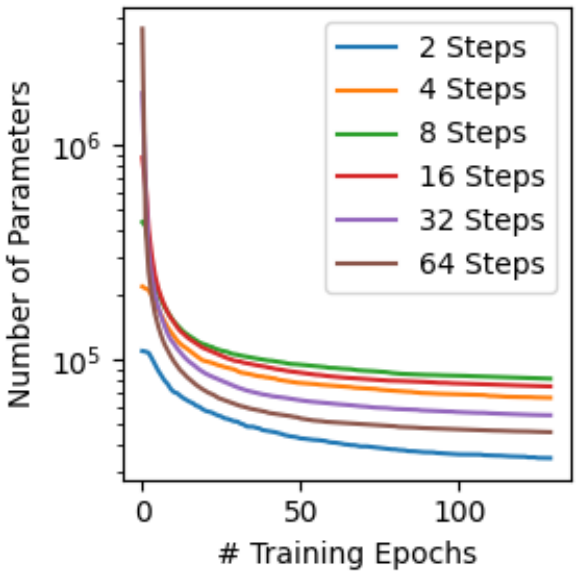

Figure 2. Models with more flow steps may end with fewer parameters, because the deeper model is expressive enough to eliminate many nodes from the intermediate layers.

iterations, but seems to have a smaller effect as the network approaches convergence. Initialization of the gate variables g has a larger impact on the initial iterations, as (for a fixed learning rate) it takes more updates to reach 0 for a gate value that is initialized close to 1. But over the course of training, the overwhelming majority of the gate variables approach zero, and then some of them stabilize at some positive value. These are the neurons that are most important to the network.

Mathematically, this may be a natural consequence of gradient descent. The gate value vector g_l pre-multiplies the inputs to layer l, and so each component g_l is multiplied by one row of W_l. Regrouping terms, Equation (3) can be rewritten

$$a_l = \sigma((W_l \odot (g_l \mathbb{1}^T))a_{l-1} + b_l), \tag{5}$$

which makes it clearer that the gates reparameterize the ordinary SGD updates to the layer, using $\tilde{W}_l = (\tilde{w}_{ij})_l = (g_i w_{ij})_l$.

Dropping layer notation for clarity, any gradient that tries to push any $\tilde{w}_{ij}$ away from zero will increase the magnitude of w_ij in an amount proportional to g_i, and the magnitude of g_i will increase proportional to w_ij. If the connection is vital to the network quality on the loss function, it will have consistent pressure to remain alive. As $g_i \to 0$, the signal to keep the connection alive becomes concentrated in g_i.

We also find that the final gate values are stable across many kinds of auxiliary parameter initialization, but that network

	# Parameters		NLL		BIC		AIC	
Flow Steps	Unpruned	Pruned	Unpruned	Pruned	Unpruned	Pruned	Unpruned	Pruned
2	108536	**33488**	124.47	**73.61**	2.614e+07	**1.511e+07**	2.511e+07	**1.479e+07**
4	217072	**65028**	53.58	**47.09**	1.322e+07	**1.017e+07**	1.115e+07	**9.549e+06**
8	434144	**76083**	**29.95**	33.72	1.099e+07	**7.619e+06**	6.858e+06	**6.895e+06**
16	868288	**73086**	**28.31**	29.72	1.566e+07	**6.785e+06**	7.398e+06	**6.090e+06**
32	1736576	**52174**	**27.89**	29.48	2.557e+07	**6.496e+06**	9.051e+06	**5.999e+06**
64	3473152	**45935**	**26.90**	30.14	4.537e+07	**6.556e+06**	1.233e+07	**6.119e+06**

Table 1. Model quality and number of parameters at various architecture complexities, evaluated on a held-out data set. Pruned networks are smaller, and while they are not always better on raw NLL, they always dominate on the Bayesian Information Criterion (BIC) and Akaike Information Criterion (AIC). Lower is better in all columns.

quality is slightly higher if the auxiliary parameters are initialized close to the same value. In the reported experiments we draw initial values from $s \sim \text{Uniform}(0.49, 0.51)$ for all s.

Finally, the prior work using stochastic gates (Louizos et al., 2017b) included a temperature parameter for the hard concrete distribution, which affects how quickly the auxiliary parameters can move relative to the network weights. We experimented with a similar parameter for the uniform stochastic and deterministic gating protocols, but found it to be unnecessary for good performance.

4. Future Work

In this work, we have developed a differentiable, smooth, in-training process for structurally reducing neural networks. This relies on a regularization constant, λ, and in our experiments $\lambda = 1$ is quite useful. Structurally, the desire to reduce the number of parameters while preserving model quality is inherently related to the Bayesian Information Criterion and Akaike Information Criterion; since these two penalize the number of parameters quite differently, they have different asymptotic qualities, but it fits with our empirical results that networks with regularized auxiliary gate parameters perform well across a wide range of λ values. Prior work (Louizos et al., 2017b) suggests that adjusting λ per layer, so that the penalty changes with the number of affected network weights tied to the neuron, may be an effective balancing strategy to improve model compression. Also, further model compression may be unnecessary once a certain model size is achieved, so lowering λ over the course of training may be useful if the network benefits from additional training of the weights after the required compression is achieved. Since network weights have a harder time learning when the gate values are close to zero, it may speed learning to periodically re-set the gate values, folding their current values into the network weights. The best practices for setting and adjusting λ, and balancing learning network weights against learning gate functions, are likely to be task-dependent, but certainly merit a more thorough investigation. Our proposed algorithm works well out of the box, but there may be a lot of performance left on the table by using such a simple system.

One of the largest demands for model compression is actually in natural language processing, where pretrained transformer models with millions (or hundreds of millions) of parameters are often adapted to domain-specific tasks with a fine-tuning round. Each layer of the transformer architectures employs a number of heads - analogous to the number of channels in the convolutional neural networks tested in this work. Evidence suggests that, after training, some of these heads may be superfluous; reducing and eliminating the number of heads during training would allow the large models to automatically compress to just the size needed for the final task. Implementing and testing our approach on these models appears to have a high potential for impact in real-world deployments.

In our experiments, we compare against only the pruning system in the literature closest to our own architecture (Louizos et al., 2017b); it would be extremely beneficial to compare against alternative approaches to model compression that are similar in spirit(Srinivas et al., 2017; Xiao & Wang, 2019; Neill et al., 2022). We would also like to test against iterative training-and-pruning approaches. Distillation (**?**) is a commonly-employed approach to learning compressed models, and we may find that the distillation objective helps the network maintain higher performance as its size decreases.

Several useful experiments are suggested by the literature on sparsification of weight matrices, and particularly the lottery ticket hypothesis (Frankle & Carbin, 2018). The compressed networks learned with regularization of auxiliary variables are clearly tied to the initialization, but we have not yet carried out any study of whether the compressed networks can solve the task as effectively if the system is retrained. If retraining the smaller network from scratch produces a very similar quality model, especially if it does so with a new random weight initialization, then it suggests that our compressed network is simply a good minimal version of

the larger model. But the lottery ticket hypothesis suggests that using larger networks at initialization improves the likelihood of finding high-quality sub-networks, and we suspect that our system performance comes, at least in part, from starting off as a much larger network.

References

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Dua, D. and Graff, C. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Fu, Y., Yu, Q., Zhang, Y., Wu, S., Ouyang, X., Cox, D., and Lin, Y. Drawing robust scratch tickets: Subnetworks with inborn robustness are found within randomly initialized networks. *Advances in Neural Information Processing Systems*, 34, 2021.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015b.

Karnin, E. D. A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks*, 1(2):239–242, 1990.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *Advances in neural information processing systems*, pp. 598–605, 1990.

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

Louizos, C., Ullrich, K., and Welling, M. Bayesian compression for deep learning. *arXiv preprint arXiv:1705.08665*, 2017a.

Louizos, C., Welling, M., and Kingma, D. P. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017b.

Malach, E., Yehudai, G., Shalev-Schwartz, S., and Shamir, O. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pp. 6682–6691. PMLR, 2020.

Meek, C., Thiesson, B., and Heckerman, D. The learning curve method applied to clustering. In *International Workshop on Artificial Intelligence and Statistics*, pp. 196–202. PMLR, 2001.

Mozer, M. C. and Smolensky, P. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in neural information processing systems*, pp. 107–115, 1989.

Neath, A. A. and Cavanaugh, J. E. The bayesian information criterion: background, derivation, and applications. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(2): 199–203, 2012.

Neill, J. O., Dutta, S., and Assem, H. Self-distilled pruning of neural networks, 2022. URL https://openreview.net/forum?id=NE8B5RQkau.

Neklyudov, K., Molchanov, D., Ashukha, A., and Vetrov, D. Structured bayesian pruning via log-normal multiplicative noise. *arXiv preprint arXiv:1705.07283*, 2017.

Ramanujan, V., Wortsman, M., Kembhavi, A., Farhadi, A., and Rastegari, M. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11893–11902, 2020.

Reed, R. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

Roy, S., Panda, P., Srinivasan, G., and Raghunathan, A. Pruning filters while training for efficiently optimizing deep learning networks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7. IEEE, 2020.

Sakamoto, Y., Ishiguro, M., and Kitagawa, G. Akaike information criterion statistics. *Dordrecht, The Netherlands: D. Reidel*, 81(10.5555):26853, 1986.

Srinivas, S., Subramanya, A., and Venkatesh Babu, R. Training sparse neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 138–145, 2017.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29:2074–2082, 2016.

Xiao, X. and Wang, Z. Autoprune: Automatic network pruning by regularizing auxiliary parameters. *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, 32, 2019.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.