Learning Neural Network Architectures using Backpropagation

Suraj Srinivas
surajsrinivas@grads.cds.iisc.ac.in
R. Venkatesh Babu
venky@cds.iisc.ac.in

Department of Computational and Data Sciences
Indian Institute of Science
Bangalore, India

Abstract

Deep neural networks with millions of parameters are at the heart of many state of the art machine learning models today. However, recent works have shown that models with much smaller number of parameters can also perform just as well. In this work, we introduce the problem of architecture-learning, i.e; learning the architecture of a neural network along with weights. We start with a large neural network, and then learn which neurons to prune. To this end, we introduce a new trainable parameter called the *Tri-State ReLU*, which helps in pruning unnecessary neurons. We also propose a smooth regularizer which encourages the total number of neurons after elimination to be small. The resulting objective is differentiable and simple to optimize. We experimentally validate our method on both small and large networks, and show that it can learn models with considerably smaller number of parameters without affecting prediction accuracy.

1 Introduction

Everything should be made as simple as possible, but not simpler - Einstein

For large-scale tasks like image classification, the general practice in recent times has been to train large networks with many millions of parameters (see [12, 19, 24]). Looking at these models, it is natural to ask - are so many parameters really needed for good performance? In other words, are these models as simple as they can be? A smaller model has the advantage of being faster to evaluate and easier to store - both of which are crucial for real-time and embedded applications. In this work, we consider the problem of automatically building smaller networks that achieve performance levels similar to larger networks.

Regularizers are often used to encourage learning simpler models. These usually restrict the magnitude (ℓ_2) or the sparsity (ℓ_1) of weights. However, to restrict the computational complexity of neural networks, we need a regularizer which restricts the width and depth of network. Here, *width* of a layer refers to the number of neurons in that layer, while *depth* simply corresponds to the total number of layers. Generally speaking, the greater the width and depth, the more are the number of neurons, the more computationally complex the model is. Naturally, one would want to restrict the total number of neurons as a means of controlling the computational complexity of the model. However, the number of neurons is an integer, making it difficult to optimize over. This work aims at making this problem easier to solve.

The overall contributions of the paper are as follows.

Pages 104.1-104.11
DOI: https://dx.doi.org/10.5244/C.30.104

- We propose novel trainable parameters which are used to restrict the total number of neurons in a neural network model - thus effectively selecting width and depth (Section 2)
- We perform experimental analysis of our method to analyze the behaviour of our method. (Section 4)
- We use our method to perform architecture selection and learn models with considerably small number of parameters (Section 4)

2 Complexity as a regularizer

In general, the term 'architecture' of a neural network can refer to aspects of a network other than width and depth (like filter size, stride, etc). However, here we use that word to simply mean width and depth. Given that we want to reduce the complexity of the model, let us formally define our notions of complexity and architecture.

Notation. *Let $\Phi = [n_1, n_2, ..., n_m, 0, 0, ...]$ be an infinite-dimensional vector whose first m components are positive integers, while the rest are zeros. This represents an m-layer neural network architecture with n_i neurons for the i^{th} layer. We call Φ as the architecture of a neural network.*

For these vectors, we define an associated norm which corresponds to our notion of architectural complexity of the neural network. Our notion of complexity is simply the total number of neurons in the network.

The true measure of computational complexity of a neural network would be the total number of weights or parameters. However, if we consider a single layer neural network, this is proportional to the number of neurons in the hidden layer. Even though this equivalence breaks down for multi-layered neural networks, we nevertheless use the same for want of simplicity.

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

Our overall objective can hence be stated as the following optimization problem.

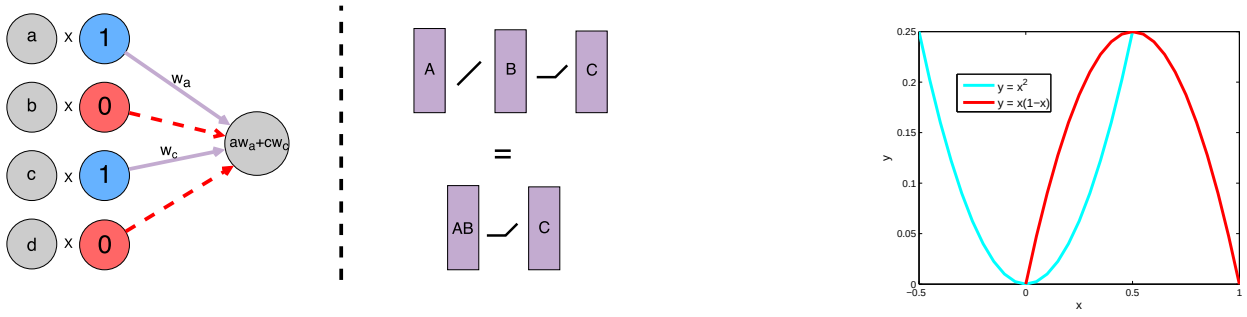
$$\hat{\theta}, \hat{\Phi} = \arg\min_{\theta, \Phi} \ell(\hat{y}(\theta, \Phi), y) + \lambda \|\Phi\| \tag{1}$$

where θ denotes the weights of the neural network, and Φ the architecture. $\ell(\hat{y}(\theta, \Phi), y)$ denotes the loss function, which depends on the underlying task to be solved. For example, squared-error loss functions are generally used for regression problems and cross-entropy loss for classification. In this objective, there exists the classical trade-off between model complexity and loss, which is handled by the λ parameter. Note that we learn both the weights (θ) as well as the architecture (Φ) in this problem. We term any algorithm which solves the above problem as an *Architecture-Learning (AL)* algorithm.

We observe that the task defined above is very difficult to solve, primarily because $\|\Phi\|$ is an integer. This makes it an integer programming problem. Hence, we cannot use gradient-based techniques to optimize for this. The main contribution of this work is the re-formulation of this optimization problem so that Stochastic Gradient Descent (SGD) and back-propagation may be used.

2.1 A Strategy for a trainable regularizer

We require a strategy to automatically select a neural network's architecture, i.e; the width of each layer and depth of the network. One way to select for width of a layer is to introduce additional learnable parameters which multiply with every neuron's output, as shown in Figure 1(a). If these new parameters are restricted to be binary, then those neurons with a zero-parameter can simply be removed. In the figure, the trainable parameters corresponding to neurons with values b and d are zero, nullifying their contribution. Thus, the sum of these binary trainable parameters will be equal to the effective width of the network. For convolutional layers with n feature map outputs, we have n additional parameters that select a subset of the n feature maps. A single additional parameter multiplies with an entire feature map either making it zero or preserving it. After all, filters are analogous to neurons for convolutional layers.



(a) (b)

Figure 1: **(a)** Our strategy for selecting width and depth. Left: Grey blobs denote neurons, coloured blobs denote the proposed additional trainable parameters. Right: Purple bars denote weight-matrices. **(b)** Graph of the ℓ_2 regularizer and the binarizing regularizer in 1-D.

To further reduce the complexity of network, we also strive to reduce the network's depth. It is well known that two neural network layers without any non-linearity between them is equivalent to a single layer, whose parameters are given by the matrix product of the weight matrices of the original two layers. This is shown on the right of Figure 1(a). We can therefore consider a trainable non-linearity, which prefers 'linearity' over 'non-linearity'. Wherever linearity is selected, the corresponding layer can be combined with the next layer. Hence, the total complexity of the neural network would be the number of parameters in layers with a non-linearity.

In this work, we combine both these intuitive observations into one single framework. This is captured in our definition of the *tri-state ReLU* which follows.

2.1.1 Definition: Tri-state ReLU

We define a new trainable non-linearity which we call the tri-state ReLU (tsReLU) as follows:

$$tsReLU(x) = \begin{cases} wx, & x \geq 0 \\ wdx, & otherwise \end{cases} \tag{2}$$

This reduces to the usual ReLU for $w = 1$ and $d = 0$. For a fixed $w = 1$ and a trainable d, this turns into parametric ReLU [9]. For us, both w and d are trainable. However, we restrict both these parameters to take only binary values. As a result, three possible states exist for this function. For $w = 0$, this function is always returns zero. For $w = 1$ and $d = 0$ it behaves similar to ReLU, while for $w = d = 1$ it reduces to the identity function.

Here, parameter w selects for the **width** of the layer, while d decides **depth**. While the w parameter is different across channels of a layer, the d parameter is tied to the same value across all channels. If $d = 1$, we can combine that layer with the next to yield a single layer. If $w = 0$ for any channel, we can simply remove that neuron as well as the corresponding weights in the next layer.

Thus, our objective while using the tri-state ReLU is

$$\begin{aligned} \underset{\theta, w_{ij}, d_i : \forall i,j}{\text{Minimize}} \quad & \ell(\hat{y}(\theta, \mathbf{w}, \mathbf{d}), y) \\ \text{such that} \quad & w_{ij}, d_i \in \{0, 1\} \\ & \forall i, j \end{aligned} \tag{3}$$

We remind the reader that here i denotes the layer number, while j denotes the j^{th} neuron in a layer. Note that for $\lambda = 0$, it converts the objective in Equation 1 from an integer programming problem to that of binary programming.

2.1.2 Learning binary parameters

Given the definition of tri-state ReLU (tsReLU) above, we require a method to learn binary parameters for w and d. To this end, we use a regularizer given by $w \times (1 - w)$ [17]. This regularizer encourages binary values for parameters, if they are constrained to lie in $[0, 1]$.

Henceforth, we shall refer to this as the *binarizing* regularizer. Murray and Ng [17] showed that this regularizer does indeed converge to binary values given a large number of iterations. For the 1-D case, this function is an downward-facing parabola with minima at 0 and 1, as shown in Figure 1(b). As a result, weights "fall" to 0 or 1 at convergence. In contrast, the ℓ_2 regularizer is an upward facing parabola with a minimum at 0, which causes it to push weights to be close to zero.

With this intuition, we now state our tsReLU optimization objective.

$$\theta, \mathbf{w}, \mathbf{d} = \underset{\theta, w_{ij}, d_i : \forall i,j}{\arg\min} \quad \ell(\hat{y}(\theta, \mathbf{w}, \mathbf{d}), y) + \lambda_1 \sum_{i=1}^{m} \sum_{j=1}^{n_i} w_{ij}(1 - w_{ij}) + \lambda_2 \sum_{i=1}^{m} d_i(1 - d_i) \tag{4}$$

Note that λ_1 is the regularization constant for the width-limiting term, while λ_2 is for the depth-limiting term. This objective can be solved using the usual back-propagation algorithm. As indicated earlier, this binarizing regularizer works only if w's and d's are guaranteed to be in $[0, 1]$. To enforce the same, we perform clipping after parameter update.

After optimization, even though the final parameters are expected to be close to binary, they are still real numbers close to 0 or 1. Let w_{ij} be the parameter obtained during the optimization. The tsReLU function uses a binarized version of this variable

$$w'_{ij} = \begin{cases} 1, & w_{ij} \geq 0.5 \\ 0, & otherwise \end{cases}$$

during the feedforward stage. Note that w_{ij} slowly changes during training, while w'_{ij} only reflects the changes made to w_{ij}. A similar equation holds for d'_i.

2.2 Adding model complexity

So far, we have considered the problem of solving Equation 1 with $\lambda = 0$. As a result, the objective function described above does not necessarily select for smaller models. Let $h_i = \sum_{j=1}^{n_i} w_{ij}$ correspond to the complexity of a layer. The model complexity term is given by

$$\|\Phi\| = \sum_{i=1}^{m} h_i \, \mathbb{1}_{(d_i=0)}$$

This is formulated such that for $d_i = 0$, the complexity in a layer is just $h_i h_{i+1}$, while for $d_i = 1$ (non-linearity absent), the complexity is 0. Overall, it counts the total number of weights in the model at convergence.

We now add a regularizer analogous to model complexity (defined above) in our optimization objective in Equation 4 . Let us call the regularizer corresponding to model complexity as $R_m(\mathbf{h}, \mathbf{d})$, which is given by

$$R_m(\mathbf{h}, \mathbf{d}) = \lambda_3 \sum_{i=1}^{m} h_i \mathbb{1}_{(d_i<0.5)} - \lambda_4 \sum_{i=1}^{m} d_i \tag{5}$$

The first term in the above equation limits the complexity of each layer's width, while the second term limits the network's depth by encouraging linearity. Note that the first term becomes zero when a non-linearity is absent. Also note that the indicator function in the first term is non-differentiable. As a result, we simply treat that term as a constant with respect to d_i.

3 Related Work

There have been many works which look at performing compression of a neural network. Weight-pruning techniques were popularized by *Optimal Brain Damage* [14] and *Optimal Brain Surgery* [8]. Recently, [21] proposed a neuron pruning technique, which relied on neuronal similarity. Our work, on the other hand, performs neuron pruning based on learning, rather than hand-crafted rules. Our learning objective can thus be seen as performing pruning and learning together, unlike the work of Han *et al.* [7], who perform both operations alternately.

Learning neural network architecture has also been explored to some extent. The *Cascade-correlation* [5] proposed a novel learning rule to 'grow' the neural network. However, it was shown for only a single layer network and is hence not clear how to scale to large deep networks. Our work is inspired from the recent work of Kulkarni *et al.* [13] who proposed to learn the width of neural networks in a way similar to ours. Specifically, they proposed to learn a diagonal matrix D along with neurons Wx, such that DWx represents that layer's neurons. However, instead of imposing a binary constraint (like ours), they learn real-weights and impose an ℓ_1-based sparsity-inducing regularizer on D to encourage zeros. By imposing a binary constraint, we are able to directly regularize for the model complexity. Recently, Bayesian Optimization-based algorithms [20] have also been proposed for automatically learning hyper-parameters of neural networks. However, for the purpose of selecting architecture, these typically require training multiple models with different architectures - while our method selects the architecture in a single run. A large number of evolutionary algorithms (see [22, 23, 26]) also exist for the task of finding Neural Network architectures.

However, these are typically evaluated on small scale problems, often not relating to pattern recognition tasks.

Many methods have been proposed to train models that are deep, yet have a lower parameterisation than conventional networks. Collins and Kohli [2] propose a sparsity inducing regulariser for backpropogation which promotes many weights to have zero magnitude. They achieve reduction in memory consumption when compared to traditionally trained models. In contrast, our method promotes *neurons* to have a zero-magnitude. As a result, our overall objective function is much simpler to solve. Denil *et al.* [3] demonstrate 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.* [25] 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.

Some recent works have also focussed on using approximations of weight matrices to perform compression. Jaderberg *et al.* [10] and Denton *et al.* [4] use SVD-based low rank approximations of the weight matrix. Gong *et al.* [6] use a clustering-based product quantization approach to build an indexing scheme that reduces the space occupied by the matrix on disk.

4 Experiments

In this section, we perform experiments to analyse the behaviour of our method. In the first set of experiments, we evaluate performance on the MNIST dataset. Later, we look at a case study on ILSVRC 2012 dataset. Our experiments are performed using the Theano Deep Learning Framework [1].

4.1 Compression performance

We evaluate our method on the MNIST dataset, using a LeNet-like [15] architecture. The network consists of two 5×5 convolutional layers with 20 and 50 filters, and two fully connected layers with 500 and 10 (output layer) neurons. We use this architecture as a starting point to learn smaller architectures. First, we learn using our additional parameters and regularizers. Second, we remove neurons with zero gate values and collapse depth for linearities wherever is it advantageous. For example, it might not be advantageous to remove depth in a bottleneck layer (like in auto-encoders). Thus, the second part of the process is human-guided.

Starting from a baseline architecture, we learn smaller architectures with variations of our method. Note that there is max-pooling applied after each of the convolutional layers, which rules out depth selection for those two layers. We compare the proposed method against baselines of directly training a neural network (NN) on the final architecture, and our method of learning a fixed final width (FFW) for various layers. In Table 1, the *Layers Learnt* column has binary elements (w,d) which denotes whether width(w) or depth(d) are learnt for each layer in the baseline network. As an example, the second row shows a method where only the *width* is learnt in the first two layers, and *depth* also learnt in the third layer. This table shows that all considered models - large and small - perform more or less equally well in terms of accuracy. This empirically shows that the small models discovered by AL preserve accuracy.

Method	λ_3	Layers Learnt	Architecture	AL (%)	NN (%)
Baseline	N/A	(0,x)-(0,x)-(0,0)	20-50-500-10	N/A	99.3
AL_1	$0.4\lambda_1$	(1,x)-(1,x)-(1,1)	16-26-10	99.07	99.08
AL_2	$0.4\lambda_1$	(1,x)-(1,x)-(1,0)	20-50-20-10	99.07	99.14
AL_3	$0.2\lambda_1$	(1,x)-(1,x)-(1,1)	16-40-10	99.22	99.25
AL_4	$0.2\lambda_1$	(1,x)-(1,x)-(1,0)	20-50-70-10	99.19	99.21

Table 1: Architecture learning performance of our method on a LeNet-like baseline. The *Layers Learnt* column has binary elements (w,d) which denotes whether width(w) or depth(d) are learnt for each layer in the baseline network. AL = Architecture Learning, NN = Neural Network trained w/o AL

We also compare the compression performance of our AL method against SVD-based compression of the weight matrix in Table 2. Here we only compress layer 3 (which has 800×500 weights) using SVD. The results show that learning a smaller network is beneficial over learning a large network and then performing SVD-based compression.

Method	Params	Accuracy (%)
Baseline	431K	99.3
SVD (rank-10)	43.6K	98.47
AL_2	**40.9K**	**99.07**
SVD (rank-40)	83.1K	99.06
AL_4	**82.3K**	**99.19**

Table 2: Comparison of compression performance of proposed method against SVD-based weight-matrix compression.

4.2 Analysis

We now perform a few more experiments to further analyse the behaviour of our method. In all cases, we train 'AL_2'-like models, and consider the third layer for evaluation. We start learning with the baseline architecture considered above.

First, we look at the effects of using different hyper-parameters . From Figure 2(a) , we observe that (i) increasing λ_3 encourages the method to prune more, and (ii) decreasing λ_1 encourages the method to learn the architecture for an extended amount of time. In both cases, we see that the architecture stays more-or-less constant after a large enough number of iterations.

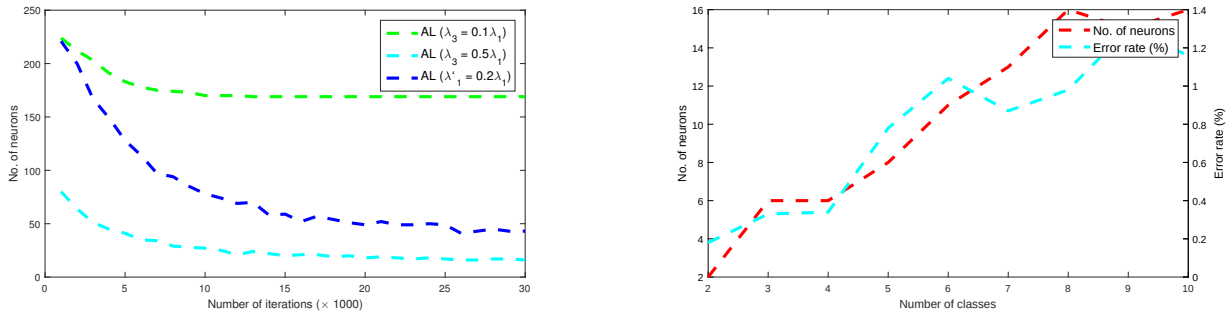
Second, we look at the learnt architectures for different amounts of data complexity. Intuitively, simpler data should lead to smaller architectures. A simple way to obtain data of differing complexity is to simply vary the number of classes in a multi-class problem like MNIST. We hence vary the number of classes from $2-10$, and run our method for each case without changing any hyper-parameters. As seen in Figure 2(b) , we see an almost monotonic increase in both architectural complexity and error rate, which confirms our hypothesis.

Third, we look at the depth-selection capabilities of our method. We used models with various initial depths and observed the depths of the resultant models. We used an initial architecture of 20 - 50 - (75 $\times$ n) - 10, where layers with width 75 are repeated to obtain a network of desired depth. We see that for small changes in the initial depth, the final learnt depth stays more or less constant. [1]

[1] For theoretical analysis of our method, please see the Supplementary material.

Initial Depth	Final Depth	Learnt Architecture	Error (%)
6	5	18-31-32-24-10	1.02
8	6	17-37-39-29-21-10	0.99
10	6	17-34-32-21-21-10	0.97
12	6	18-34-30-21-17-10	1.04
15	8	16-37-35-25-20-22-10	0.93

Table 3: Performance of the proposed method on networks of increasing depth.



(a) (b)

Figure 2: **(a)** Plot of the architecture learnt against the number of iterations. We see that λ_1 affects convergence rate while λ_3 affects amount of pruning.
(b) Plot of the no. of neurons learnt for MNIST with various number of classes. We see that both the neuron count and the error rate increase with increase in number of classes.

4.3 Architecture Selection

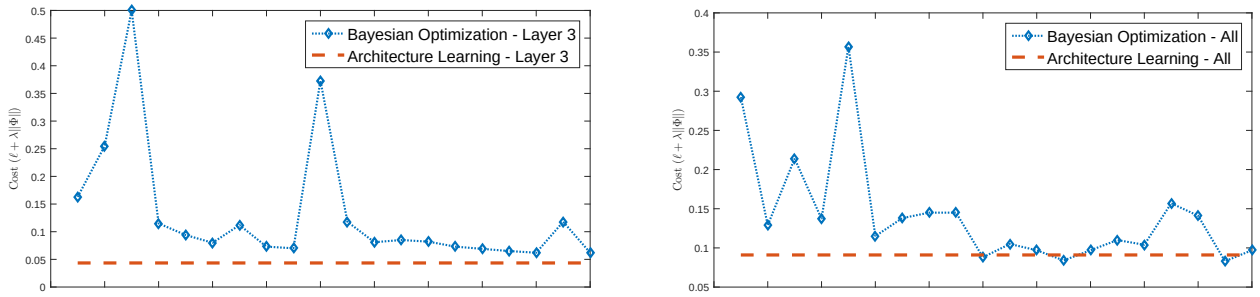
In recent times, Bayesian Optimization (BO) has emerged as a compelling option for hyper-parameter optimization. In these set of experiments, we compare the architecture-selection capabilities of our method against BO. In particular, we use the Spearmint-lite software package [20] with default parameters for our experiments.

We use BO to first determine the width of the last FC layer (a single scalar), and later, the width of all three layers (3 scalars). For comparison, we use the same objective function for both BO and Architecture-Learning. This means that we use $\lambda_3 = 10^{-5}$ for AL, while we externally compute the cost after every training run for BO. Figure 3 shows that BO typically needs multiple runs to discover networks which perform close to AL. Performing such multiple runs is often prohibitive for large networks. Even for a small network like ours, training took $\sim$30 minutes on a TitanX GPU for 300 epochs. Training with AL does not change the training time, whereas using BO we spent $\sim$10 hours for completing 20 runs. Further, AL directly optimizes the cost function as opposed to BO, which performs a black-box optimization.

Given that we perform architecture selection, what hyper-parameters does AL need? We notice that we only need to decide four quantities - λ_{1-4}. If our objective is to only decide widths, we need to decide only two quantities - λ_1 and λ_3. Thus, for a n-layer neural network, we are able to decide n (or $2n-1$) numbers (widths and depths) based on only two (or four) global hyper-parameters. In the Appendix, we shall look at heuristics for setting these hyper-parameters.

4.4 Case study: AlexNet

For the experiments that follow, we use an AlexNet-like [12] model, called CaffeNet, provided with the Caffe Deep Learning framework. It is very similar to AlexNet, except that the order of max-pooling and normalization have been interchanged. We use the ILSVRC 2012 [18] validation set to compute accuracies in the Table 4. Unlike the experiments performed



(a) (b)

Figure 3: **(a)** Comparison against Bayesian Optimization for the case of learning the width of only third layer. **(b)** Similar comparison for learning the widths of all three layers. ($\lambda = 10^{-5}$ in both cases)

previously, we start with a pre-trained model and then perform architecture learning (AL) on the learnt weights. We see that our method performs almost as well as the state of the art compression methods. This means that one can simply use a smaller neural network instead of using weight re-parameterization techniques (FastFood, SVD) on a large network.

Further, many compression methods are formulated keeping only fully-connected layers in mind. For tasks like Semantic Segmentation, networks with only convolutional layers are used [16]. Our results show that the proposed method can successfully prune both fully connected neurons and convolutional filters. Further, ours (along with SVD) is among the *few* compression methods that can utilize dense matrix computations, whereas all other methods require specialized kernels for sparse matrix computations [7] or custom implementations for diagonal matrix multiplication [25], etc.

Method	Params	Accuracy (%)	Compression (%)
Reference Model (CaffeNet)	60.9M	57.41	0
Neuron Pruning ([21])	39.6M	55.60	35
SVD-quarter-F ([25])	25.6M	56.19	58
Adaptive FastFood 16 ([25])	18.7M	57.10	69
AL-conv-fc	19.6M	55.90	68
AL-fc	19.8M	54.30	68
AL-conv	47.8M	55.87	22

Table 4: Compression performance on CaffeNet.

Method	Layers Learnt	Architecture
Baseline	N/A	96 256 384 384 256 4096 4096 1000
AL-fc	fc[6,7]	96 256 384 384 256 1536 1317 1000
AL-conv	conv[1,2,3,4,5]	80 127 264 274 183 4096 4096 1000
AL-conv-fc	conv[5] - fc[6,7]	96 256 384 384 237 1761 1661 1000

Table 5: Architectures learnt by our method whose performance is given in Table 4.

5 Conclusions

We have presented a method to learn a neural network's architecture along with weights. Rather than directly selecting width and depth of networks, we introduced a small number of real-valued hyper-parameters which selected width and depth for us. We also saw that we get smaller architectures for MNIST and ImageNet datasets that perform on par with the large architectures. Our method is very simple and straightforward, and can be suitably applied to any neural network. This can also be used as a tool to further explore the dependence of architecture on the optimization and convergence of neural networks.

References

[1] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[2] Maxwell D. Collins and Pushmeet Kohli. Memory bounded deep convolutional networks. *CoRR*, abs/1412.1442, 2014. URL http://arxiv.org/abs/1412.1442.

[3] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

[4] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[5] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. 1989.

[6] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[7] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

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

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[10] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference*. BMVA Press, 2014.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[13] Praveen Kulkarni, Joaquin Zepeda, Frederic Jurie, Patrick PÃl'rez, and Louis Chevallier. Learning the structure of deep architectures using l1 regularization. In Mark W. Jones Xianghua Xie and Gary K. L. Tam, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 23.1–23.11. BMVA Press, September 2015.

ISBN 1-901725-53-7. doi: 10.5244/C.29.23. URL https://dx.doi.org/10.5244/C.29.23.

[14] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, 1989.

[15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[16] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

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

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 2015. doi: 10.1007/s11263-015-0816-y.

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

[20] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[21] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In Mark W. Jones Xianghua Xie and Gary K. L. Tam, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 31.1–31.12. BMVA Press, September 2015. ISBN 1-901725-53-7. doi: 10.5244/C.29.31. URL https://dx.doi.org/10.5244/C.29.31.

[22] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[23] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[25] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. *arXiv preprint arXiv:1412.7149*, 2014.

[26] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.