Learning Compact Architectures for Deep Neural Networks

A THESIS
SUBMITTED FOR THE DEGREE OF
Master of Science (Engineering)
IN THE
Faculty of Engineering

BY

Suraj Srinivas

Department of Computational and Data Science
Indian Institute of Science
Bangalore – 560 012 (INDIA)

January, 2017

Declaration of Originality

I, **Suraj Srinivas**, with SR No. **06-02-00-10-21-14-1-11462** hereby declare that the material presented in the thesis titled

Learning Compact Architectures for Deep Neural Networks

represents original work carried out by me in the **Deparment of Computational and Data Sciences** at **Indian Institute of Science** during the years **2014-2016**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

© Suraj Srinivas
January, 2017
All rights reserved

Acknowledgements

Of all the people who have helped me during the course of my Master's degree for the past two years, I would first like to thank my advisor Dr. R. Venkatesh Babu. His deep intuition of the subject matter, along with his penchant for empiricism have fundamentally shaped my views on how research should be done.

My interest in Machine Learning stems from my wonderful experience in the course offered by Dr. Shivani Agarwal. Her clarity of thought and presentation, combined with her enthusiasm for the learning problem have had a profound impact on my research.

My time in IISc was memorable largely due to my association with the Video Analytics Lab. I would like to thank all current and former lab-members, especially Reddy, Lokesh, Akshay, Ravi, Srinivas, Nikita, Nithish, Shiv, Swami and Ram, for the endless stimulating discussions.

I am forever deeply indebted to my parents and my younger sister for their encouragement and support throughout these years. They made sure I had nothing other than my research to worry about.

Finally, the campus of IISc has featured as a prominent backdrop to my time here. I would like to thank the Institute for providing and maintaining a peaceful green campus and an atmosphere conducive to research.

Abstract

Deep Neural Networks (NNs) have recently emerged as the model of choice for a wide a range of Machine Learning applications ranging from computer vision to speech recognition to natural language processing and beyond. However, these models are also known for being large and cumbersome, limiting their applicability to real-time and embedded applications. As a result, it is desirable to *compress* these models to make them easier to store and process.

Training Neural Networks is often described as a kind of 'black magic', as successful training requires setting the right hyper-parameter values (such as the number of neurons in a layer, depth of the network, etc). It is often not clear what these values should be, and these decisions often end up being either ad-hoc or driven through extensive experimentation. It would be desirable to automatically set some of these hyper-parameters for the user so as to minimize trial-and-error. Combining this objective with our earlier preference for smaller models, we ask the following question - for a given task, is it possible to come up small neural network architectures automatically?

In this thesis, we propose methods to achieve the same. Our contributions here are four-fold. First, we describe a method that takes a pre-trained network model and performs compression without using training data. Second, we perform compression and network training at the same time, in a data-driven manner. We call this method 'Architecture-Learning'. Third, we show connections of Architecture-Learning with a popular regularizer called 'Dropout', and propose the so-called 'Generalized Dropout' regularizer. Lastly, we apply the Architecture-Learning methodology to sparsify neural networks, i.e.; remove weights to create sparse weight matrices. Our methods are competitive with state-of-the-art methods that perform network compression, and as well as those that perform architecture selection.

Publications based on this Thesis

1. Suraj Srinivas and R. Venkatesh Babu, "Data-free Parameter Pruning for Deep Neural Networks". British Machine Vision Conference (BMVC) 2015 [45]

2. Suraj Srinivas and R. Venkatesh Babu, "Learning Neural Network Architectures using Backpropagation". British Machine Vision Conference (BMVC) 2016 [46]

3. Suraj Srinivas, Ravi Kiran Sarvadevabhatla, Konda Reddy Mopuri, Nikita Prabhu, Srinivas SS Kruthiventi, and R. Venkatesh Babu. "A Taxonomy of Deep Convolutional Neural Nets for Computer Vision". Frontiers in Robotics and AI 2 (2016): 36. [48]

4. Suraj Srinivas and R. Venkatesh Babu, "Generalized Dropout", arXiv pre-print, arXiv:1611.06791 [47]

5. Suraj Srinivas, Akshay Varun and R. Venkatesh Babu, "Training Sparse Neural Networks", arXiv pre-print, arXiv:1611.06694 [49]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The human brain is surprisingly small for the amount of sophisticated computation it performs. Attempts at simulating brain activities usually require massive supercomputers that need several kilowatts of power. This has been a constant feature of most Machine Learning models as well - models that perform well are often unusually large. This holds true even for the case of deep neural networks, where typical models are often over-parametrized to facilitate easier training. This paradigm of using large models has been largely fuelled by the availability of massively parallel GPUs (Graphics Processing Units). Without GPUs, training time would increase from a few days to a few months, making it infeasible to train large models with million of parameters. Further, inference time would also increase from a fraction of a second to a few seconds. This makes it difficult to deploy large models on small embedded devices and have real-time inference. This reliance of Deep Neural Networks on GPUs is deeply unsatisfactory.

We can tackle this problem in two ways. The first approach is to build specialized hardware suited to Deep Neural Networks. This can take advantage of Neural Network structure to achieve low power and fast computation. Alternately, we can build *smaller* models with similar performance and a smaller computational load, so that these models can now run efficiently on the available embedded hardware. In this thesis, we broadly look at the second approach. In modern Deep Learning literature, this is also referred to as the problem of *Model Compression*.

Given our premise, one can naturally ask - if smaller models can work just as well as large ones, wouldn't researchers and engineers use them to begin with? While this is indeed true, unfortunately it is not easy to identify which of these small models perform well. In other words, it is difficult to search over the space of possible architectures to determine a good candidate architecture. Once a suitable neural network structure is identified, it is relatively simple to train that network and achieve good performance. This is indeed one of the broad goals of this thesis - to identify small architectures which work just as well as large ones.

In this Introductory chapter, we provide some background material which serve as context for the thesis. This chapter is divided into two parts. First, we briefly introduce Deep Neural Networks and the computational aspects associated with them. Second, we look at some possible ways to re-parameterize Neural Networks so that it is easy to identify small, but well-performing models.

1.1 Computational Aspects of Deep Neural Networks

Convolutional Neural Networks (CNNs) have largely taken over Computer Vision in recent times. In this section, we introduce these CNNs from a computational perspective. We assume that the reader is largely familiar with traditional neural networks, which we shall call "fully connected layers" in this thesis. The content in this section is largely adapted from a part of our survey article on CNNs for vision [48].

The idea of a Convolutional Neural Network (CNN) is not new. These models had been shown to work well for hand-written digit recognition [32]. However, due to the inability of these networks to scale to much larger images, they slowly fell out of favour. This was largely due to memory and hardware constraints, and the unavailability of large amounts of training data. With increase in computational power thanks to wide availability of GPUs, and the introduction of large scale datasets like the ImageNet (see [41]) and the MIT Places dataset (see [55]), it was possible to train larger, more complex models. This was first shown by the popular *AlexNet* model which was discussed earlier. This largely kick-started the usage of deep networks in computer vision. Figure 1.1 shows a representation of the weights in the *AlexNet* model. While the first five layers are convolutional, the last three are fully connected layers.

1.1.1 Fully Connected Layers

Traditional neural network layers perform matrix multiplication on a vector x to produce another vector $y = f(Wx + b)$, where W is a matrix and b is a scalar, also called the bias. Here, $f(\cdot)$ is an elementwise non-linearity. We refer to these layers as fully connected layers. If x is the input data, then y is generally called the vector of activations or neural activations. The length of the vector y is usually referred to as the *width* of that layer. Each element in W is a *weight*, while an entire row is typically called a *neuron*.

Neurons are generally more interpretable than weights. While individual weights simply denote the 'strength of connection' between the input and output, neurons represent basis functions or features extracted by the network.

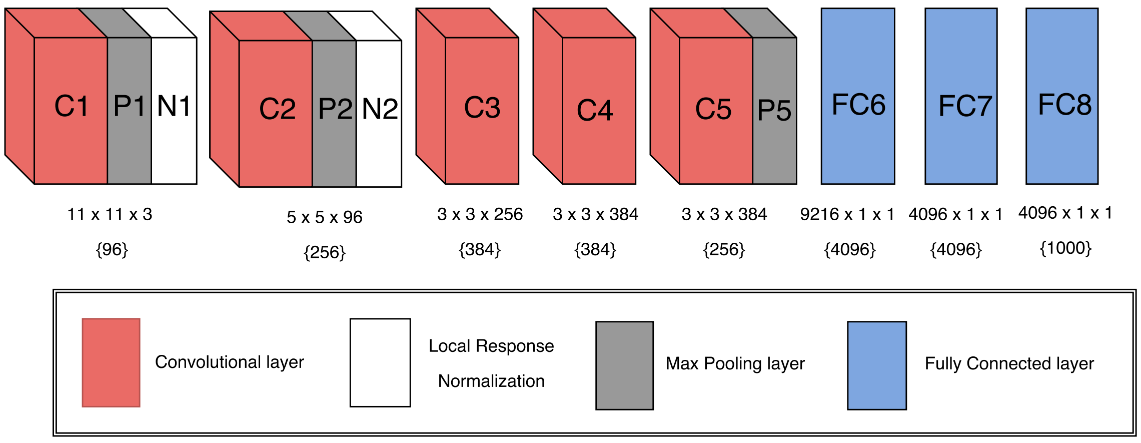


Figure 1.1: An illustration of the weights in the AlexNet model. Note that after every layer, there is an implicit ReLU non-linearity. The number inside curly braces represents the number of filters with dimensions mentioned above it.

1.1.2 Convolutional Layers

Using traditional neural networks for real-world image classification is impractical for the following reason: Consider a 2D image of size 200×200 for which we would have have $40,000$ input nodes. If the hidden layer has $20,000$ nodes, the size of the matrix of input weights would be $40,000 \times 20,000 = 800$ Million. This is just for the first layer – as we increase the number of layers, this number increases even more rapidly. Besides, vectorizing an image completely ignores the complex 2D spatial structure of the image. How do we build a system that overcomes both these disadvantages?

One way is to use 2D convolutions instead of matrix multiplications. Learning a set of convolutional filters (each of 11×11, say) is much more tractable than learning a large matrix (40000×20000). 2D convolutions also naturally take the 2D structure of images into account. Alternately, convolutions can also be thought of as regular neural networks with two constraints (See [6]):

- Local connectivity: This comes from the fact that we use a convolutional filter with dimensions much smaller than the image it operates on. This contrasts with the *global* connectivity paradigm typically relevant to vectorized images.

- Weight sharing: This comes from the fact that we perform convolutions, i.e. we apply the same filter across the image. This means that we use the same *local* filters on many locations in the image. In other words, the weights between all these filters are shared.

In practical CNNs however, the convolution operations are not applied in the traditional sense wherein the filter shifts one position to the right after each multiplication. Instead, it is common to use larger shifts (commonly referred to as stride). This is equivalent to performing image down-sampling after regular convolution.

If we wish to train these networks on RGB images, one would need to learn multiple *multi-channel* filters. In the representation in Figure 1.1, the numbers $11 \times 11 \times 3$, along with $\{96\}$ below **C1** indicates that there are 96 filters in the first layers, each of spatial dimension of 11×11, with one for each of the 3 RGB channels.

We note that this paradigm of convolution like operations (location independent feature-detectors) is not entirely suitable for registered images. As an example, images of faces require different feature-detectors at different spatial locations. To account for this, [52] consider only locally-connected networks with no weight-sharing. Thus, the choice of layer connectivity depends on the underlying type of problem.

In modern deep neural networks such as AlexNet, convolutional layers typically perform more number of operations compared to fully connected layers. This is primarily because convolutions need to be performed over large images - which requires a large amount of multiply-add operations. However, these layers are typically parameter efficient, as the size of convolutional kernels are very small (11×11) compared to the image size (224×224). In contrast, fully connected layers are usually applied at a stage where significant dimensionality reduction is already done. As a result, these layers end up being relatively fast to compute. However, because of the inefficiency of such generic layers, they contain much larger number of parameters when compared to convolutional layers. This is shown in Table 1.1.

	Inference Speed	**Number of parameters**
Convolutional Layer	Slow	Low
Fully Connected Layer	Fast	High

Table 1.1: Computational demands of convolutional layers vs fully connected layers in modern neural networks.

1.1.3 Non-linearities

Deep networks usually consist of convolutions followed by a non-linear operation after each layer. This is necessary because cascading linear systems (like convolutions) is another linear system. Non-linearities between layers ensure that the model is more expressive than a linear model.

Traditional feedforward neural networks used the sigmoid ($\sigma(x) = \frac{1}{1+e^{-x}}$) or the tanh ($\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$) non-linearities. However, modern convolutional networks use the ReLU ($\text{ReLU}(x) = max(0, x)$) non-linearity. CNNs with this non-linearity have been found to train faster, as shown by [37]. In addition, these non-linearities require negligible amount of computation, as these are simply element-wise operations.

Another kind of non-linearity typically used in deep networks is that of pooling, which is used in conjunction with convolutional layers. The most common form of pooling is max-pooling. This operation can be thought of as a *max filter*, where each $n \times n$ region is replaced with it's max value. This operation serves two purposes:

1. It picks out the highest activation in a local region, thereby providing a small degree of spatial invariance. This is analogous to the operation of complex cells.

2. It reduces the size of the activation for the next layer by a factor of n^2. With a smaller activation size, we need a smaller number of parameters to be learnt in the later layers.

When designing neural networks, placement of these max-pooling layers are often critical. If too much max pooling is used, performance suffers, while too little a max pooling results in an explosion of the number of parameters.

1.1.4 Why are CNNs large?

Modern Deep Networks such as AlexNet [29] and VGGNet [42] contain 60 Million and 138 Million parameters respectively. This takes up a lot of storage space - 220+ and 500+ MB respectively. As pointed out in Table 1.1, most of these parameters lie in fully-connected (FC) layers. As a result, it is natural to ask - are large FC layers really necessary? Indeed, GoogleNet [51] uses much smaller FC layers (and cleverly-designed 'inception' layers) to avoid parameter explosion.

In general, architecture design decisions are often arbitrary - they are mainly decided keeping in mind computational considerations rather than concerns regarding parsimony of representation. Large, over-parameterized neural networks have been found to generalize better for large datasets. Given that CNNs take a long time of train, it is lucrative to use large networks to obtain good results immediately. However, this mode of working is deeply unsatisfactory - it is unclear where there exists smaller CNNs that work just as well. This is the primary focus of this thesis - to design methods that automatically discover such small CNNs.

1.2 Compressing Neural Networks

We shall now describe some popular ways to compress deep neural networks so that one can optimize for either the parameter count or the number of operations.

1.2.1 Distillation

The term *Model Compression* was first used by Bucilua *et al.* [8] back in 2006. They described a method to compress a large ensemble of models (not necessarily neural networks) to a single compact model that can be used at test time. A similar idea was explored by Hinton *et al.* [24] for neural network compression, who used the term *Knowledge Distillation* (KD). The overall idea behind KD is to leverge the so-called *knowledge* acquired by training a large network to train a smaller network. It has been shown that this is advantageous to having the smaller model learn from scratch.

Neural Networks compute class probabilites using the *softmax* function, which is defined as follows.

$$q_i = \frac{exp(z_i)}{\sum_i exp(z_i)} \tag{1.1}$$

Here, z_i denotes the i^{th} output class of the neural network in a classification problem, and q_i denotes the softmax output. This softmax output is always between 0 and 1, denoting a probability distribution over classes. Typically, one computes cross-entropy error of the true label with the softmax output for training the network.

However, one problem with the softmax output distribution $q = \{q_1, q_2, ...q_i, ...\}$ is that it usually places a lot of mass (close to 1) to one of the q_i's and close to zero mass to all other q's. In other words, softmax is a 'soft' way of choosing the maximum of many numbers. This transformation of z's into q's preserves only the label information and not much else. We require a transform that provides additional information about z as well. To facilitate this, we use the following transform.

$$q_i = \frac{exp(z_i/T)}{\sum_i exp(z_i/T)} \tag{1.2}$$

Here, T is a scalar value called 'Temperature'. Using $T > 1$ produces a softer probability distribution over classes than regular softmax. For example, if the input to an image classification network is the image of a car, the softmax outputs peak at 'car' and are close to zero for all other classes. However, if we consider the temperature softened softmax, the probability q_i is maximum for 'car', but it is also high for related classes such as 'truck', 'train', while being

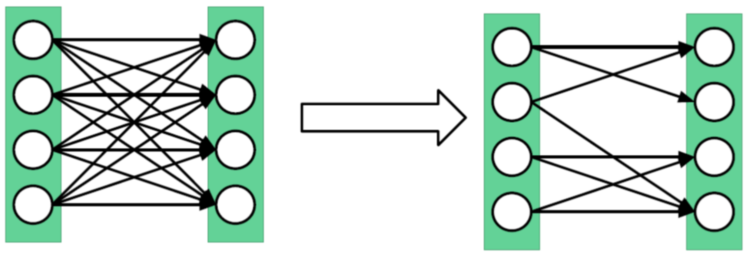


Figure 1.2: An Illustration of weight removal / sparsification in a Neural Network matrix

low for unrelated classes like ‘dog’ and ‘cat’. The idea behind Knowledge Distillation is to use such softened labels from a large network for training a smaller network.

To perform Knowledge Distillation, we first train a large network. Let it’s temperature softened softmax outputs be Q_l^T. Let the softened output for the untrained smaller network be Q_s^T. Let the usual softmax output for smaller network be denoted by Q_s. We now train the smaller network with the following loss function.

$$Loss = \lambda_1 \ell(Q_s, Y) + \lambda_2 \ell(Q_s^T, Q_l^T) \tag{1.3}$$

Here, $\ell(\cdot)$ denotes the cross-entropy loss function. Note that here we use both the regular softmax loss (first term) and the softened hints provided by the larger network (second term).

The advantage of this method is that it is applicable to any two neural networks, irrespective of their architectures. However, a practical disadvantage of this method is that this behaviour has not been shown to scale up to larger networks trained on Imagenet-like datasets.

1.2.2 Sparsification

Neural Networks typically consist of operations such as dense convolutions and dense matrix multiplications. Would using sparse matrix multiplications and convolutions be more efficient? This is the objective of sparsifying neural networks, shown in Figure 1.2.

To use these, we require that majority of the parameters or weights in a neural network be zero. To encourage this, Collins and Kohli [11] use sparsity inducing priors on the weight matrix. This is done by projecting onto ℓ_0 balls after each gradient computation. This is identical to performing the so-called *projected gradient descent* with the constraint on the ℓ_0 norm of weights. In practice, this is done by repeatedly hard-thresholding all but the top-t weights (in terms of magnitude) after each iteration.

Han *et al.* [18] use a similar strategy. However, instead of a monolithic training step, they alternate between training and pruning. That is, pruning is not performed at every iteration, but is performed only once after several epochs. Using this strategy, they prune AlexNet by

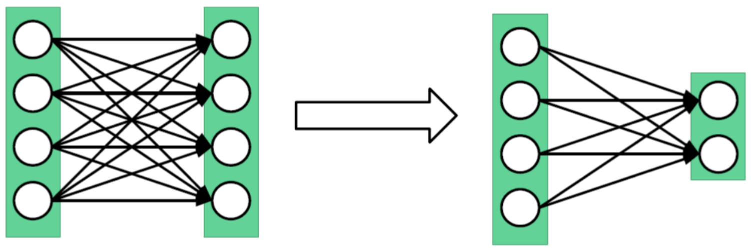


Figure 1.3: Removing Neurons in a Neural Network

upto 9× and obtain 3× speedup. This method is currently among the state-of-the-art methods for minimizing parameter count.

One disadvantage of this approach is that practical implementation requires support for sparse matrix multiplications and convolutions. Further, these provide significant speedups only when the sparsity level is sufficiently high ($> 90\%$).

1.2.2.1 Special Case: Block Sparsification

Rather than dropping arbitrary weights, we may instead choose to drop a set of weights that constitute a neuron / feature detector, as shown in Figure 1.3. This method is advantageous over the earlier method in that this does not require support for sparse matrix multiplication or convolution. In other words, we retain the canonical structure of the neural network while reducing the model size. However, this method has a disadvantage in that significantly lower number of weights can be pruned because of the extra constraint of removing a whole set. In this thesis, we mainly look at methods to perform block sparsification.

1.2.3 Quantization

Weights in neural networks are usually stored as 32-bit floating point numbers. However, some recent works have found that reduced precision can also be used for inference. In the same spirit, there emerged many works which performed clever non-linear quantization of the weights. However, these are very difficult to implement in general purpose hardware. Here we shall briefly discuss the special case of binary neural networks, which are easier to deploy.

1.2.3.1 Binary Neural Nets

There exist several attempts at creating binary neural nets, however, most of them do not scale to large datasets. Here, we shall briefly discuss 'XNOR-Net' [39], which has been shown to work for large CNNs such as AlexNet.

Here, we shall only attempt explaining the method for fully-connected layers, as the method for convolutional layers is slight more involved. First, the paper introduces a method to obtain

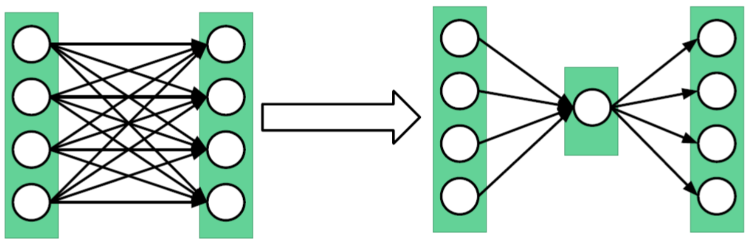


Figure 1.4: An illustration of matrix factorization applied to a Neural Network layer

binary weights. Given a weight matrix W, we obtain a binary matrix B by optimizing the following.

$$\mathbf{B}^*, \alpha^* = \text{argmin}_{B,\alpha} \|\mathbf{W} - \alpha \mathbf{B}\|^2 \tag{1.4}$$

The solution to this is to simply set $B = sign(W)$ and $\alpha = \frac{\|W\|_1}{\|W\|_0}$. Similarly, to binarize both weights and activations, we use the following objective function.

$$\mathbf{B}^*, \mathbf{H}^*, \alpha^*, \beta^* = \text{argmin}_{B,H,\alpha,\beta} \|\mathbf{W^T X} - \alpha\beta \mathbf{B^T H}\|^2 \tag{1.5}$$

Here, H is the binarized version of X, the real-valued input variable. This objective function is solved by decomposing it into two binarization problems, each of the form of Equation 1.4.

Using this form of binarization, Rastegari *et al.* [39] report 58× increase in inference speed with a 32× compression. However, they suffer a 12% loss in accuracy over the AlexNet baseline.

1.2.4 Factorization

If we assume that neural network weights are redundant, it is reasonable to expect them to have a low-rank structure. In case of fully connected layers, it was observed [54] that such a low-rank structure indeed exists. By re-arranging the matrix in terms of it's singular vectors, it is possible to create a bottleneck layer with very few parameters.

Kim *et al.* [27] make the astute observation that similar to using matrix decompositions for fully connected layers, one can similarly use tensor decompositions to compress convolutional layers.

There are two popular ways to perform tensor decomposition:

- CP or CANDECOMP/PARAFAC decomposition: This extends the notion of spectral decomposition of matrices. For tensors, this decomposes a tensor into sum of 'k' rank-1 tensors, where 'k' is equivalent to the rank of the tensor.

- Tucker decomposition: Here the tensor is broken into a series of matrices and one small 'core' matrix.

In terms of practical implementation, Tucker decomposition is much more attractive as it is more stable. The decomposition is done so that we re-use structures which are already accelerated by low-level GPU libraries.

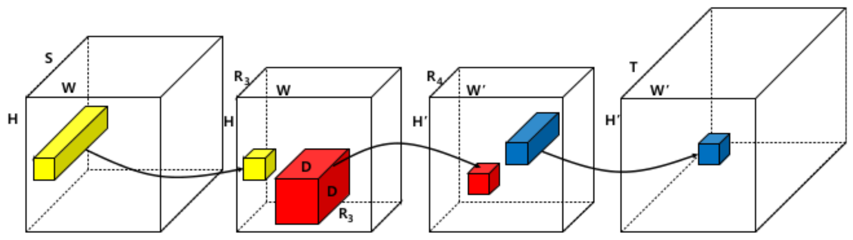


Figure 1.5: A schematic diagram of tensor decomposition. Image Credits: Kim *et al.* [27]

In particular, Kim *et al.* [27] use a Tucker-2 decomposition, which involves decomposing a 4-tensor into 2 matrices and one small 'core' tensor. Simply put, the first layer decreases the number of channels and the last layer increases it to the original number. The second 'core' tensor in effect performs convolution with reduced number of input and output filters. This is summarized by figure 1.5.

The advantages of this method is identical to that of neuron pruning / block sparsification - this preserves the canonical structure of the neural network, and does not require any specialized matrix operations to be implemented. However, one potential disadvantage of this method over block sparsification is that it artificially increases the number of processing stages, or the apparent depth, which can potentially increase inference time.

1.3 Outline of the thesis

The main focus on this thesis is to develop methods that perform block sparsification. As this task keeps the canonical structure of neural network intact, this can also be thought of as selecting the number of neurons in a layer. The thesis is organized as follows.

1. **Data-free Neuron Pruning:** Chapter 2 begins our investigation into neural network redundancy and asks the question - is it possible to remove neurons from a network without having access to the data it was trained on? The answer, surprisingly, is yes. We show that this pruning can happen without significantly affecting predictive performance. This exposes the significant redundancy that exists in neural networks after training.

2. **Architecture Learning using Backpropagation:** In Chapter 3, we design a data-driven method to remove neurons. This is integrated as a part of the training process. We call this method '*Architecture Learning*', as this learns both the architecture and weights in a single optimization framework. Experiments show that simply finding the smallest architecture is competitive with more sophisticated model compression methods.

3. **Connection with Dropout:** The Architecture Learning algorithm is reminiscent of Dropout - in both cases neurons are removed in the training process. Does there exist a deeper relationship between these two approaches? Chapter 4 answers this question in the affirmative. We consider a stochastic version of Architecture Learning called *Stochastic Architecture Learning* (SAL). We show that SAL and Dropout belong to a much larger family of regularizers which we call *Generalized Dropout*. Other members of this family include adaptive versions of Dropout, which we call *Dropout++*. These regularizers are derived using a Bayesian framework based on Variational Inference.

4. **Application on Training Sparse Networks:** We apply the methods developed in the previous chapters to the task of removing weights in a neural network rather than neurons. Chapter 5 contains the details of this method. In addition to this, we also make connections of our method to *Spike-and-Slab* priors, and stochastic optimization. Experiments show that this method is competitive with state-of-the-art weight pruning methods.

Chapter 2

Data-free Neuron Pruning

I have made this letter longer than usual, only because I have not had the time to make it shorter [1] - Blaise Pascal

2.1 Introduction

Aspiring writers are often given the following advice: produce a first draft, then *remove* unnecessary words and *shorten* phrases whenever possible. Can a similar recipe be followed while building deep networks? For large-scale tasks like object classification, the general practice [29, 42, 51] has been to use large networks with powerful regularizers [50]. This implies that the overall model complexity is much smaller than the number of model parameters. 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.

Given such a large network, how do we make it smaller? A naive approach would be to remove weights which are close to zero. However, this intuitive idea does not seem to be theoretically well-founded. LeCun *et al.* proposed *Optimal Brain Damage* (OBD) [31], a theoretically sound technique which they showed to work better than the naive approach. A few years later, Hassibi *et al.* came up with *Optimal Brain Surgeon* (OBS) [19], which was shown to perform much better than OBD, but was much more computationally intensive. This line of work focusses on pruning unnecessary weights in a trained model.

There has been another line of work in which a smaller network is trained to mimic a much larger network. Bucila *et al.* [8] proposed a way to achieve the same - and trained smaller models which had accuracies similar to larger networks. Ba and Caruna [2] used the approach to show that shallower (but much wider) models can be trained to perform as well as deep

[1]Loosely translated from French

models. Knowledge Distillation (KD) [24] is a more general approach, of which Bucila *et al.* 's is a special case. FitNets [40] use KD at several layers to learn networks which are deeper but thinner (in contrast to Ba and Caruna's shallow and wide), and achieve high levels of compression on trained models.

Many methods have been proposed to train models that are deep, yet have a lower parameterisation than conventional networks. Collins and Kohli [11] 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. Denil *et al.* [12] 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. Ciresan *et al.* [10] train networks with random connectivity, and show that they are more computationally efficient than densely connected networks.

Some recent works have focussed on using approximations of weight matrices to perform compression. Jenderberg *et al.* [25] and Denton *et al.* [13] use SVD-based low rank approximations of the weight matrix. Gong *et al.* [16], on the other hand, use a clustering-based product quantization approach to build an indexing scheme that reduces the space occupied by the matrix on disk. Unlike the methods discussed previously, these do not need any training data to perform compression. However, they change the network structure in a way that prevents operations like fine-tuning to be done easily after compression. One would need to 'uncompress' the network, fine-tune and then compress it again.

Similar to the methods discussed in the paragraph above, our pruning method doesn't need any training/validation data to perform compression. Unlike these methods, our method merely prunes parameters, which ensures that the network's overall structure remains same - enabling operations like fine-tuning on the fly. The following section explains this in more detail.

2.2 Wiring similar neurons together

Given the fact that neural nets have many redundant parameters, how would the weights configure themselves to express such redundancy? In other words, when can weights be removed from a neural network, such that the removal has no effect on the net's accuracy?

Suppose that there are weights which are exactly equal to zero. It is trivial to see that these can be removed from the network without any effect whatsoever. This was the motivation for the naive magnitude-based removal approach discussed earlier.

In this work we look at another form of redundancy. Let us consider a toy example of a NN with a single hidden layer, and a single output neuron. This is shown in figure 2.1. Let $W_1, W_2, ... \in \mathcal{R}^d$ be vectors of weights (or 'weight-sets') which includes the bias terms, and

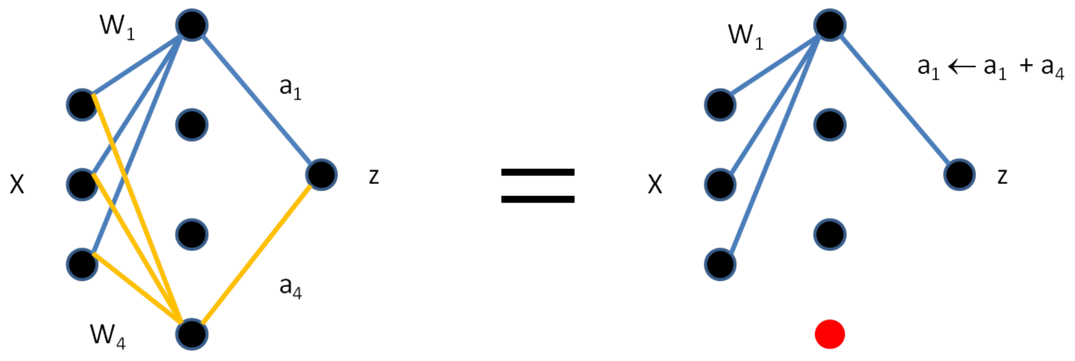


Figure 2.1: A toy example showing the effect of equal weight-sets ($W_1 = W_4$). The circles in the diagram are neurons and the lines represent weights. Weights of the same colour in the input layer constitute a weight-set.

$a_1, a_2, ... \in \mathcal{R}$ be scalar weights in the next layer. Let $X \in \mathcal{R}^d$ denote the input, with the bias term absorbed. The output is given by

$$z = a_1 h(W_1^T X) + a_2 h(W_2^T X) + a_3 h(W_3^T X) + ... + a_n h(W_n^T X) \tag{2.1}$$

where $h(\cdot)$ is a monotonically increasing non-linearity, such as sigmoid or ReLU.

Now let us suppose that $W_1 = W_2$. This means that $h(W_1^T X) = h(W_2^T X)$. Replacing W_2 by W_1 in (2.1), we get

$$z = (a_1 + a_2) h(W_1^T X) + 0\ h(W_2^T X) + a_3 h(W_3^T X) + ... + a_n h(W_n^T X)$$

This means whenever two *weight sets* (W_1, W_2) are equal, one of them can effectively be removed. Note that we need to alter the co-efficient a_1 to $a_1 + a_2$ in order to achieve this. We shall call this the '*surgery*' step. This reduction also resonates with the well-known *Hebbian* principle, which roughly states that "neurons which fire together, wire together". If we find neurons that fire together ($W_1 = W_2$), we wire them together ($a_1 = a_1 + a_2$). Hence we see here that along with single weights being equal to zero, equal weight vectors also contribute to redundancies in a NN. Note that this approach assumes that the same non-linearity $h(\cdot)$ is used for all neurons in a layer.

2.3 The case of dissimilar neurons

Using the intuition presented in the previous section, let us try to formally derive a process to eliminate neurons in a trained network. We note that two weight sets may never be exactly equal in a NN. What do we do when $\|W_1 - W_2\| = \|\epsilon_{1,2}\| \geq 0$? Here $\epsilon_{i,j} = W_i - W_j \in \mathcal{R}^d$.

As in the previous example, let z_n be the output neuron when there are n hidden neurons. Let us consider two similar weight sets W_i and W_j in z_n and that we have chosen to remove W_j to give us z_{n-1}.

We know that the following is true.

$$z_n = a_1 h(W_1^T X) + ... + a_i h(W_i^T X) + ... + a_j h(W_j^T X) + ...$$

$$z_{n-1} = a_1 h(W_1^T X) + ... + (a_i + a_j) h(W_i^T X) + ...$$

If $W_i = W_j$ (or $\epsilon_{i,j} = 0$), we would have $z_n = z_{n-1}$. However, since $\|\epsilon_{i,j}\| \geq 0$, this need not hold true. Computing the squared difference $(z_n - z_{n-1})^2$, we have

$$(z_n - z_{n-1})^2 = a_j^2 (h(W_j^T X) - h(W_i^T X))^2 \tag{2.2}$$

To perform further simplification, we use the following Lemma.

Lemma 1 *Let $a, b \in \Re$ and $h(\cdot)$ be a monotonically increasing function, such that max $\left(\frac{\mathrm{d}h(x)}{\mathrm{d}x}\right) \leq 1, \forall x \in \Re$. Then,*

$$(h(a) - h(b))^2 \leq (a - b)^2$$

The proof for this is provided in the Appendix. Note that non-linearities like sigmoid and ReLU [29] satisfy the above property. Using the Lemma and (2.2), we have

$$(z_n - z_{n-1})^2 \leq a_j^2 \ (\epsilon_{i,j}^T X)^2$$

This can be further simplified using Cauchy-Schwarz inequality.

$$(z_n - z_{n-1})^2 \leq a_j^2 \ \|\epsilon_{i,j}\|_2^2 \ \|X\|_2^2$$

Now, let us take expectation over the random variable X on both sides. Here, X is assumed to belong to the input distribution represented by the training data.

$$E(z_n - z_{n-1})^2 \leq a_j^2 \ \|\epsilon_{i,j}\|_2^2 \ E\|X\|_2^2$$

Note that $E\|X\|_2^2$ is a scalar quantity, independent of the network architecture. Given the above expression, we ask which (i, j) pair least changes the output activation. To answer this, we take minimum over (i, j) on both sides, yielding

$$min(E(z_n - z_{n-1})^2) \leq min(a_j^2 \; \|\epsilon_{i,j}\|_2^2) \; E\|X\|_2^2 \tag{2.3}$$

To minimize an *upper bound* on the expected value of the squared difference, we thus need to find indicies (i, j) such that $a_j^2 \; \|\epsilon_{i,j}\|_2^2$ is the least. Note that we need not compute the value of $E\|X\|_2^2$ to do this - making it dataset independent. Equation (2.3) takes into consideration both the naive approach of removing near-zero weights (based on a_j^2) and the approach of removing similar weight sets (based on $\|\epsilon_{i,j}\|_2^2$).

The above analysis was done for the case of a single output neuron. It can be trivially extended to consider multiple output neurons, giving us the following equation

$$min(E\langle(z_n - z_{n-1})^2\rangle) \leq min(\langle a_j^2\rangle \; \|\epsilon_{i,j}\|_2^2) \; E\|X\|_2^2 \tag{2.4}$$

where $\langle\cdot\rangle$ denotes the average of the quantity over all output neurons. This enables us to apply this method to intermediate layers in a deep network. For convenience, we define the saliency of two weight-sets in (i, j) as $s_{i,j} = \langle a_j^2\rangle \; \|\epsilon_{i,j}\|_2^2$.

We elucidate our procedure for neuron removal here:

1. Compute the saliency $s_{i,j}$ for all possible values of (i, j). It can be stored as a square matrix M, with dimension equal to the number of neurons in the layer being considered.

2. Pick the minimum entry in the matrix. Let it's indices be (i', j'). Delete the j'^{th} neuron, and update $a_{i'} \leftarrow a_{i'} + a_{j'}$.

3. Update M by removing the j'^{th} column and row, and updating the i'^{th} column (to account for the updated $a_{i'}$.)

The most computationally intensive step in the above algorithm is the computation of the matrix M upfront. Fortunately, this needs to be done only once before the pruning starts, and only single columns are updated at the end of pruning each neuron.

2.3.1 Connection to Optimal Brain Damage

In the case of toy model considered above, with the constraint that only weights from the hidden-to-output connection be pruned, let us analyse the OBD approach.

The OBD approach looks to prune those weights which have the least effect on the training/validation error. In contrast, our approach looks to prune those weights which change the output neuron activations the least. The saliency term in OBD is $s_j = h_{jj}a_j^2/2$, where h_{ii} is the

i^{th} diagonal element of the Hessian matrix. The equivalent quantity in our case is the saliency $s_{i,j} = a_j^2 \|\epsilon_{ij}\|_2^2$. Note that both contain a_j^2. If the change in training error is proportional to change in output activation, then both methods are equivalent. However, this does not seem to hold in general. Hence it is not always necessary that the two approaches remove the same weights.

In general, OBD removes a single weight at a time, causing it to have a finer control over weight removal than our method, which removes a set of weights at once. However, we perform an additional 'surgery' step ($a_i \leftarrow a_i + a_j$) after each removal, which is missing in OBD. Moreover, for large networks which use a lot of training data, computation of the Hessian matrix (required for OBD) is very heavy. Our method provides a way to remove weights quickly.

2.3.2 Connection to Knowledge Distillation

Hinton *et al.* [24] proposed to use the 'softened' output probabilities of a learned network for training a smaller network. They showed that as $T \to \infty$, their procedure converges to the case of training using output layer neurons (without softmax). This reduces to Bucila *et al.* 's [8] method. Given a larger network's output neurons z_l and smaller network's neurons z_s, they train the smaller network so that $(z_l - z_s)^2$ is minimized.

In our case, z_l corresponds to z_n and z_s to z_{n-1}. We minimize an upper bound on $E((z_l - z_s)^2)$, whereas KD exactly minimizes $(z_l - z_s)^2$ over the training set. Moreover, in the KD case, the minimization is performed over *all* weights, whereas in our case it is only over the output layer neurons. Note that we have the expectation term (and the upper bound) because our method does not use any training data.

2.3.3 Weight normalization

In order for our method to work well, we need to ensure that we remove only those weights for which the RHS of (2.3) is small. Let $W_i = \alpha W_j$, where α is a positive constant (say 0.9). Clearly, these two weight sets compute very similar features. However, we may not be able to eliminate this pair because of the difference in magnitudes. We hence propose to normalise all weight sets while computing their similarity.

Result 1 *For the ReLU non-linearity, defined by $max(0, \cdot)$, and for any $\alpha \in \Re_+$ and any $x \in \Re$, we have the following result:*

$$max(0, \alpha x) = \alpha \; max(0, x)$$

Using this result, we scale all weight sets $(W_1, W_2, ...)$ such that their norm is one. The α

factor is multiplied with the corresponding co-efficient in the next layer. This helps us identify better weight sets to eliminate.

2.3.4 Some heuristics

While the mathematics in the previous section gives us a good way of thinking about the algorithm, we observed that certain heuristics can improve performance.

The usual practice in neural network training is to train the bias without any weight decay regularization. This causes the bias weights to have a much higher magnitude than the non-bias weights. For this reason, we normalize only the non-bias weights. We also make sure that the similarity measure ϵ takes 'sensible-sized' contributions from both weights and biases. This is accomplished for fully connected layers as follows.

Let $W = [W' \ b]$, and let $W'_{(n)}$ correspond to the normalized weights. Rather than using $\|\epsilon_{i,j}\| = \|W_i - W_j\|$, we use $\|\epsilon_{i,j}\| = \dfrac{\|W'_{(n)i} - W'_{(n)j}\|}{\|W'_i + W'_j\|} + \dfrac{|b_i - b_j|}{|b_i + b_j|}$.

Note that both are measures of similarity between weight sets. We have empirically found the new similarity measure performs much better than just using differences. We hypothesize that this could be a tighter upper bound on the quantity $E((z_n - z_{n-1})^2)$.

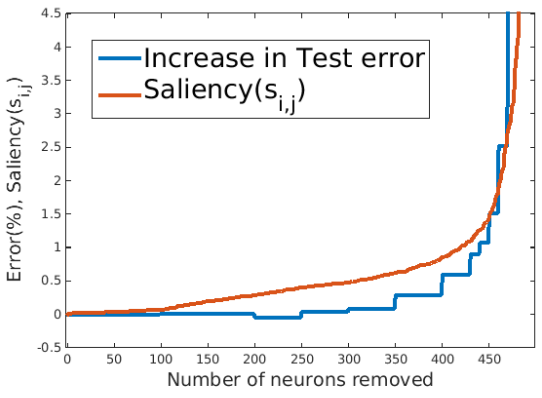
Similar heuristics can be employed for defining a similarity term for convolutional layers. In this work, however, we only consider fully connected layers.

2.4 How many neurons to remove?

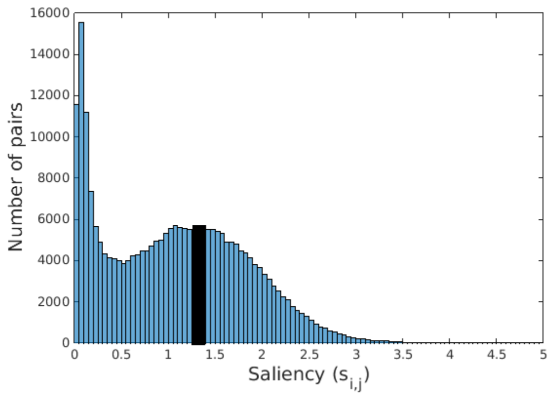
One way to use our technique would be to keep removing neurons until the test accuracy starts going below certain levels. However, this is quite laborious to do for large networks with multiple layers.

We now ask whether it is possible to somehow determine the number of removals automatically. Is there some indication given by removed weights that tell us when it is time to stop? To investigate the same, we plot the saliency $s_{i,j}$ of the removed neuron as a function of the order of removal. For example, the earlier pruned neurons would have a low value of saliency $s_{i,j}$, while the later neurons would have a higher value. The red line in Figure 2.2(a) shows the same. We observe that most values are very small, and the neurons at the very end have comparatively high values. This takes the shape of a distinct exponential-shaped curve towards the end.

One heuristic would probably be to have the cutoff point near the foot of the exponential curve. However, is it really justified? To answer the same, we also compute the increase in test error (from baseline levels) at each stage of removal (given by the blue line). We see that the error stays constant for the most part, and starts increasing rapidly near the exponential. Scaled appropriately, the saliency curve could be considered as a proxy for the increase in test error. However, computing the scale factor needs information about the test error curve. Instead, we could use the slope of saliency curve to estimate how densely we need to sample the test error. For example, fewer measurements are needed near the flatter region and more measurements are needed near the exponential region. This would be a **data-driven** way to determine number of neurons to remove.



(a)



(b)

Figure 2.2: (a) Scaled appropriately, the saliency curve closely follows that of increase in test error ; (b) The histogram of saliency values. The black bar indicates the mode of the gaussian-like curve.

We also plot the histogram of values of saliency. We see that the foot of the exponential (saliency ≈ 1.2) corresponds to the mode of the gaussian-like curve (Figure 2.2(b)). If we require a **data-free** way of finding the number of neurons to remove, we simply find the saliency value of the mode in the histogram and use that as cutoff. Experimentally, we see that this works well when the baseline accuracy is high to begin with. When it is low, we see that using this method causes a substantial decrease in accuracy of the resulting classifier. In this work, we use fractions (0.25, 0.5, etc) of the number given by the above method for large networks. We choose the best among the different pruned models based on validation data. A truly data-free method, however, would require us to not use any validation data to find the number of neurons to prune. Note that only our pruning method is data-free. The formulation of such a complete data-free method for large networks demands further investigation.

2.5 Experiments and Results

In most large scale neural networks [29, 42] , the fully connected layers contain most of the parameters in the network. As a result, reducing just the fully connected layers would considerably compress the network. We hence show experiments with only fully connected layers.

2.5.1 Comparison with OBS and OBD

Given the fact that Optimal Brain Damage/Surgery methods are very difficult to evaluate for mid-to-large size networks, we attempted to compare it against our method on a toy problem. We use the SpamBase dataset [1], which comprises of 4300 datapoints belonging to two classes, each having 57 dimensional features. We consider a small neural network architecture - with a single hidden layer composed of 20 neurons. The network used a sigmoidal non-linearity (rather than ReLU), and was trained using Stochastic Gradient Descent (SGD). The NNSYSID [1] package was used to conduct these experiments.

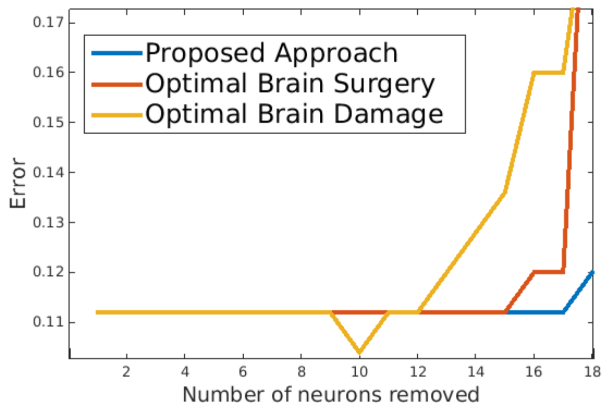


Figure 2.3: Comparison of proposed approach with OBD and OBS. Our method is able to prune many more weights than OBD/OBS at little or no increase in test error

Figure 2.3 is a plot of the test error as a function of the number of neurons removed. A 'flatter' curve indicates better performance, as this means that one can remove more weights for very little increase in test error. We see that our method is able to maintain is low test error as more weights are removed. The presence of an additional 'surgery' step in our method improves performance when compared to OBD. Figure 2.4 shows performance of our method when surgery is not performed. We see that the method breaks down completely in such a scenario. OBS performs inferior to our method because it presumably prunes away important weights early on - so that any surgery is not able to recover the original performance level. In

[1]http://www.iau.dtu.dk/research/control/nnsysid.html

addition to this, our method took < 0.1 seconds to run, whereas OBD took 7 minutes and OBS took > 5 hours. This points to the fact that our method could scale well for large networks.

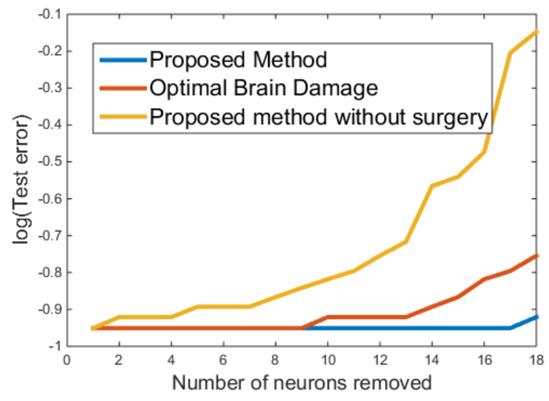


Figure 2.4: Comparison with and without surgery. Our method breaks down when surgery is not performed. Note that the y-axis is the log of test error.

2.5.2 Experiments on LeNet

We evaluate our method on the MNIST dataset, using a LeNet-like [32] architecture. This set of experiments was performed using the Caffe Deep learning framework [26]. The network consisted of a two 5×5 convolutional layers with 20 and 50 filters, and two fully connected layers with 500 and 10 (output layer) neurons. Noting the fact that the third layer contains 99% of the total weights, we perform compression only on that layer.

Neurons pruned	Naive method	Random removals	Ours	Compression (%)
150	99.05	98.63	99.09	29.81
300	98.63	97.81	98.98	59.62
400	97.60	92.07	98.47	79.54
420	96.50	91.37	98.35	83.52
440	94.32	89.25	97.99	87.45
450	92.87	86.35	97.55	89.44
470	62.06	69.82	94.18	93.47

Table 2.1: The numbers represent accuracies in (%) of the models on a test set. 'Naive method' refers to removing neurons based on magnitude of weights. The baseline model with 500 neurons had an accuracy of 99.06%. The highlighted values (in **red**) are those predicted for cutoff by our cut-off selection methods.

The results are shown in Table 2.1. We see that our method performs much better than the naive method of removing weights based on magnitude, as well as random removals - both of which are data-free techniques.

Our data-driven cutoff selection method predicts a cut-off of 420, for a 1% decrease in accuracy. The data-free method, on the other hand, predicts a cut-off of 440. We see that immediately after that point, the performance starts decreasing rapidly.

2.5.3 Experiments on AlexNet

For networks like AlexNet [29], we note that there exists two sets of fully connected layers, rather than one. We observe that pruning a given layer changes the weight-sets for the next layer. To incorporate this, we first prune weights in earlier layers before pruning weights in later layers.

For our experiments, we use an AlexNet-like architecture, 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 [41] validation set to compute accuracies in the following table.

# FC6 pruned	# FC7 pruned	Accuracy (%)	Compression (%)	# weights removed
2800	0	48.16	61.17	37M
2100	0	53.76	45.8	27.9M
1400	0	56.08	30.57	18.6M
700	0	57.68	15.28	9.3M
0	2818	49.76	23.5	14.3M
0	2113	54.16	17.6	10.7M
0	1409	56.00	11.8	7.2M
0	704	57.76	5.88	3.5M
1400	2854	44.56	47.88	29.2M
1400	2140	50.72	43.55	26.5M
1400	1427	53.92	39.22	23.9M
1400	713	55.6	34.89	21.27M

Table 2.2: Compression results for CaffeNet. The first two columns denote the number of neurons pruned in each of the FC6 and FC7 layers. The validation accuracy of the unpruned CaffeNet was found to be 57.84%. Note that it has 60.9M weights in total. The numbers in **red** denote the best performing models, and those in **blue** denote the numbers predicted by our data-free cutoff selection method.

We observe that using fractions (0.25, 0.5, 0.75) of the prediction made by our data-free method gives us competitive accuracies. We observe that removing as many as 9.3 million parameters in case of 700 removed neurons in FC6 only reduces the base accuracy by 0.2%. Our best method was able to remove upto 21.3 million weights, reducing the base accuracy by only 2.2%.

2.6 Conclusion

We proposed a data-free method to prune neurons in a Neural Network. Our method weakly relates to both Optimal Brain Damage and a form of Knowledge Distillation. By minimizing the expected squared difference of logits we were able to avoid using any training data for model compression. We also observed that the saliency curve has low values in the beginning and exponentially high values towards the end. This fact was used to decide on the number of neurons to prune. Our method can be used on top of most existing model architectures, as long as they contain fully connected layers.

Appendix

Proof: [Proof of Lemma 1] Given $h(\cdot)$ is monotonically increasing, and $max\ \left(\frac{\mathrm{d}h(x)}{\mathrm{d}x}\right) \leq 1,\ \forall\, x \in \mathfrak{R}$.

$$\implies 0 < \frac{dh(x)}{dx} \leq 1 \implies \int_b^a 0\ \mathrm{d}x < \int_b^a \mathrm{d}h(x) \leq \int_b^a \mathrm{d}x \implies 0 < h(a) - h(b) \leq a - b$$

Since both $h(a) - h(b) > 0$, and $a - b > 0$, we can square both sides of the inequality.

$$(h(a) - h(b))^2 \leq (a - b)^2$$

$\square$

Chapter 3

Architecture Learning using Backpropagation

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

3.1 Introduction

So far, we have seen that neural networks can be pruned even without using any training data. Can we prune better if we used training data? In this chapter, we consider the problem of automatically building smaller networks that achieve performance levels similar to larger networks. More importantly, this reduction is performed using training data - which is what is used to learn weights as well. As we shall soon see, this conveniently takes the form of a regularizer added to the overall loss function.

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 chapter are as follows.

- 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.

- We perform experimental analysis of our method to analyze the behaviour of our method.
- We use our method to perform architecture selection and learn models with considerably small number of parameters

3.2 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 1 *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. In Chapter 5 of this thesis, where we consider weights instead of neurons, this equivalence holds for arbitrary networks.

Definition 1 *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.

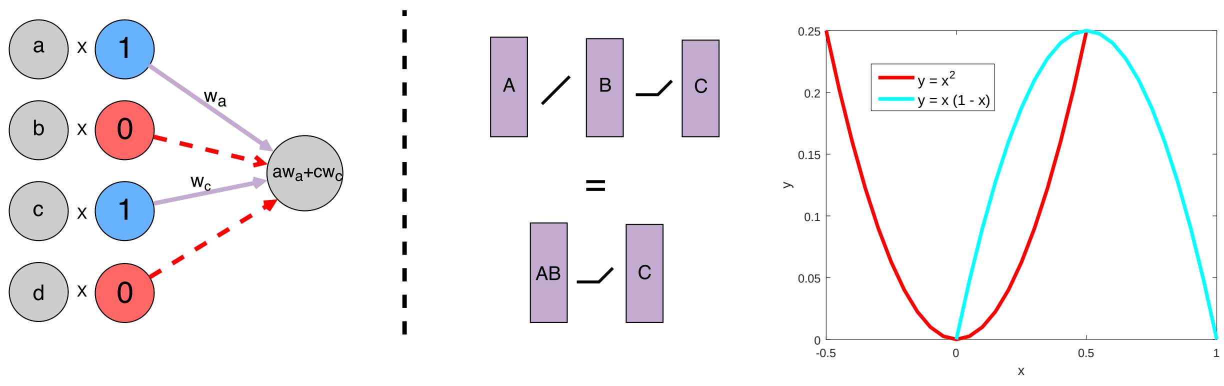
$$\hat{\theta}, \hat{\Phi} = \arg\min_{\theta, \Phi} \ \ell(\hat{y}(\theta, \Phi), y) + \lambda \|\Phi\| \tag{3.1}$$

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.

3.2.1 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 3.1(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 3.1: **(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 3.1(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.

3.2.1.1 Definition: Tri-state ReLU

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

$$tsReLU(x) = \begin{cases} wx, & x \geq 0 \\ wdx, & otherwise \end{cases} \tag{3.2}$$

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 [20]. 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

$$\begin{aligned} \underset{\theta, w_{ij}, d_i : \forall i,j}{\text{Minimize}} \quad & \ell(\hat{y}(\theta, \mathbf{w}, \mathbf{d}), y) \\ \text{such that} \quad & w_{ij}, d_i \in \{0, 1\} \ , \ \forall i, j \end{aligned} \tag{3.3}$$

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 3.1 from an integer programming problem to that of binary programming.

3.2.1.2 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 $w \times (1 - w)$ [36]. 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 [36] 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 3.1(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.

$$\theta, \mathbf{w}, \mathbf{d} \quad = \quad \underset{\theta, w_{ij}, d_i : \forall i,j}{\arg\min} \quad \ell(\hat{y}(\theta, \mathbf{w}, \mathbf{d}), y) \; + \; \lambda_1 \sum_{i=1}^{m} \sum_{j=1}^{n_i} w_{ij}(1 \; - \; w_{ij}) \; + \; \lambda_2 \sum_{i=1}^{m} d_i(1 \; - \; d_i) \qquad (3.4)$$

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.

3.2.2 Adding model complexity

So far, we have considered the problem of solving Equation 3.1 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 the i^{th} layer, obtained by adding over all n_i neurons in that 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 of a layer is just h_i, while for $d_i = 1$ (non-linearity absent), the complexity is 0. Overall, it counts the total number of neurons in the model at convergence.

We now add a regularizer analogous to model complexity (defined above) in our optimization objective in Equation 3.4 . Let us call the regularizer corresponding to model complexity as $R_m(\mathbf{h}, \mathbf{d})$, which is given by

$$R_m(\mathbf{h}, \mathbf{d}) = \lambda_3 \sum_{i=1}^{m} h_i \mathbb{1}_{(d_i<0.5)} - \lambda_4 \sum_{i=1}^{m} d_i \tag{3.5}$$

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.

3.3 Properties of the method

Here we identify a few properties of our architecture selection method.

1. **Non-redundancy of architecture**: The learnt final architecture must not have any redundant neurons. Removing neurons should necessarily degrade performance.

2. **Local-optimality of weights**: The performance of the learnt final architecture must at least be equal to a trained neural network initialized with this final architecture.

3. **Mirroring data-complexity**: A 'harder' dataset should result in a larger model than an 'easier' dataset.

We intuitively observe that all these properties would automatically hold if a 'master' property which requires both the architecture and the weights be globally optimal holds. Given that the optimization objective of neural networks is highly non-convex, global optimality cannot be guaranteed. As a result, we restrict ourselves to studying the three properties listed.

In the text that follows, we provide statements that hold for our method. These are obtained by analysing widths of each layer of a neural network assuming that depth is never collapsed. In other words, these hold for neural networks with a single hidden layer.

Non-redundancy of architecture

This is an important property that forms the main motivation for doing architecture-learning. Such a procedure can replace the node-pruning techniques that are used to compress neural networks.

Proposition 1 *At convergence, the loss (ℓ) of the proposed method over the train set satisfies* $\frac{\partial \ell}{\partial \Phi} < 0$

This statement implies that change in architecture is inversely proportional to change in loss. In other words, if the architecture grows smaller, the loss must increase. While there isn't a strict relationship between loss and accuracy, a high loss generally indicates worse accuracy.

Local Optimality of weights

The proposed method learns both architecture and weights. What would happen if we initialized a neural network with this learnt architecture, and proceeded to learn only the weights? This property ensures that in both cases we fall into a local minimum with architecture Φ.

Proposition 2 *Let ℓ_1 be the loss over the train set at convergence obtained by training a neural network on data $\mathcal{D}$ with a fixed architecture Φ. Let ℓ_2 be the loss at convergence when the neural network is trained with the proposed method on data $\mathcal{D}$ such that it results in the same final architecture Φ. Then, $\frac{\partial \ell_1}{\partial \theta} < \epsilon$ and $\frac{\partial \ell_2}{\partial \theta} < \epsilon$ for any $\epsilon \to 0$.*

Mirroring data-complexity

Characterizing data-complexity has traditionally been hard. Here, we consider the following approach.

Proposition 3 *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two datasets which produce train losses ℓ_1 and ℓ_2 upon training with a fixed architecture Φ such that $\ell_1 > \ell_2$. When trained with the proposed method, the final architectures $\hat{\Phi}_1$ and $\hat{\Phi}_2$ (corresponding to $\mathcal{D}_1$ and $\mathcal{D}_2$) satisfy the relation $\|\hat{\Phi}_1\| > \|\hat{\Phi}_2\|$ at convergence.*

Here, $\mathcal{D}_1$ is the 'harder' dataset because it produces a higher loss on the same neural network architecture. As a result, the 'harder' dataset always produces a larger final architecture. We do not provide a proof for this statement. Instead, we experimentally verify this in Section 3.5.2.

3.3.1 Proofs of Propositions

Let $E = \ell + \lambda_b \mathcal{R}_b + \lambda_m \mathcal{R}_m$ be total objective function, where $\mathcal{R}_b$ is the binarizing regularizer, $\mathcal{R}_m = \|\phi\|$ is the model complexity term. At convergence, we assume that $\mathcal{R}_b = 0$ as the corresponding weights are all binary or close to binary. Let the maximum step size (due to gradient clipping) for $\mathbf{w}$ and $\mathbf{d}$ be s.

Proof: [Proof of proposition 1] At convergence, we assume
$\frac{\partial E}{\partial \phi} < \epsilon$, for some $\epsilon \to 0_+$.

$$\frac{\partial \ell}{\partial \phi} < -\lambda_m \frac{\partial \|\phi\|}{\partial \phi} + \epsilon \implies \frac{\partial \ell}{\partial \phi} < -\lambda_m + \epsilon \implies \frac{\partial \ell}{\partial \phi} < 0$$

for some ϵ sufficiently small.

$\square$

Proof: [Proof of proposition 2] Let $\mathcal{R}_b = 0$ at t_1^{th} iteration with architecture Φ_1. Let Φ_2 be the architecture at iteration $t_2 > t_1$ such that at iterations $t_1 < t < t_2$, architecture is Φ_1.

$\implies \exists$ an iteration $t_1 < t < t_2$ such that $\mathcal{R}_b > s(1-s) = s_1$, s being the maximum step size.

Let $q = \frac{\partial \ell_2}{\partial \theta_2}$. Let λ_b be parameterized by k as follows.

$$\lambda_b \; s_1 = \mathbb{E}_{\mathcal{D}}(q) + k\sigma \quad \text{where} \quad \sigma = \mathbb{E}_{\mathcal{D}}(q - \mathbb{E}_{\mathcal{D}}(q))^2$$
$$\text{If} \quad k \to \infty \quad \text{then} \quad \mathbb{P}(q > \lambda_b \; s_1) \to 0$$

Hence, for large enough λ_b, $\Phi_2 = \Phi_1$. After $T >> t$ iterations, we have

$$\frac{\partial \ell_1}{\partial \theta_1} < \epsilon \quad \text{and} \quad \frac{\partial \ell_2}{\partial \theta_2} < \epsilon \tag{3.6}$$

for some $\epsilon \to 0_+$. However, if $\theta_1 \in \mathbb{R}^{d_1}$, then $\theta_2 \in \mathbb{R}^{d_2}$, such that $d_1 < d_2$.

Without loss of generality, let us assume that neurons corresponding to first d_1 weights are selected for, while the rest are inactive. As a result, $\frac{\partial \ell_2}{\partial \theta_2(d)} = 0$, for $d \in [d_1, d_2]$. Hence, the following holds $\frac{\partial \ell_2}{\partial \theta_1} < \epsilon$. This, along with equation 3.6, proves the assertion.

$\square$

3.4 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* [31] and *Optimal Brain Surgery* [19]. Recently, [45] 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.* [18], who perform both operations alternately.

Learning neural network architecture has also been explored to some extent. The *Cascade-correlation* [14] 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.* [30] 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 [43] 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.

Many methods have been proposed to train models that are deep, yet have a lower parameterisation than conventional networks. Collins and Kohli [11] 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.* [12] 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.* [54] 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.* [25] and Denton *et al.* [13] use SVD-based low rank approximations of the weight matrix. Gong *et al.* [16] use a clustering-based product quantization approach to build an indexing scheme that reduces the space occupied by the matrix on disk.

3.5 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 [5].

3.5.1 Compression performance

We evaluate our method on the MNIST dataset, using a LeNet-like [32] 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 3.1, 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.

We also compare the compression performance of our AL method against SVD-based compression of the weight matrix in Table 3.2. 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.

3.5.2 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 3.2(a) , we

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 3.1: 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

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

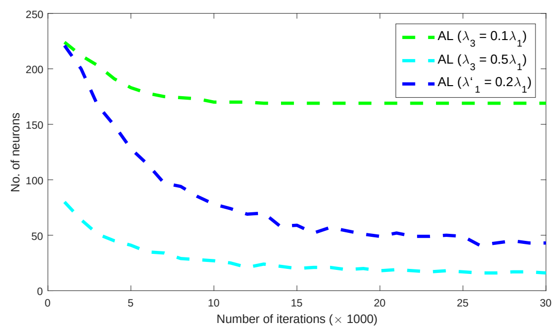
observe that (i) increasing λ_3 encourages the method to prune more, and (ii) decreasing λ_1 encourages the method to learn the architecture for an extended amount of time. In both cases, we see that the architecture stays more-or-less constant after a large enough number of iterations.

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 3.2(b) , we see an almost monotonic increase in both architectural complexity and error rate, which confirms our hypothesis.

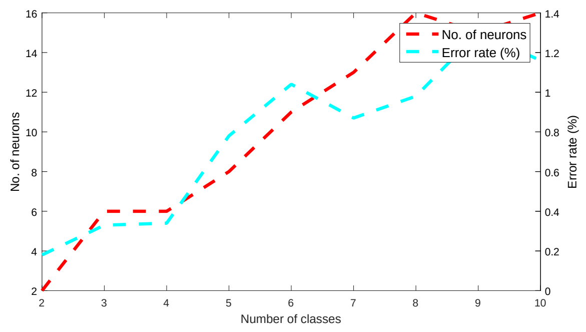
Third, we look at the depth-selection capabilities of our method. We used models with various initial depths and observed the depths of the resultant models. We used an initial architecture of 20 - 50 - (75 $\times$ n) - 10, where layers with width 75 are repeated to obtain a network of desired depth.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

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 3.3: Performance of the proposed method on networks of increasing depth.



(a)



(b)

Figure 3.2: **(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.

3.5.3 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 [43] 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 3.3 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.

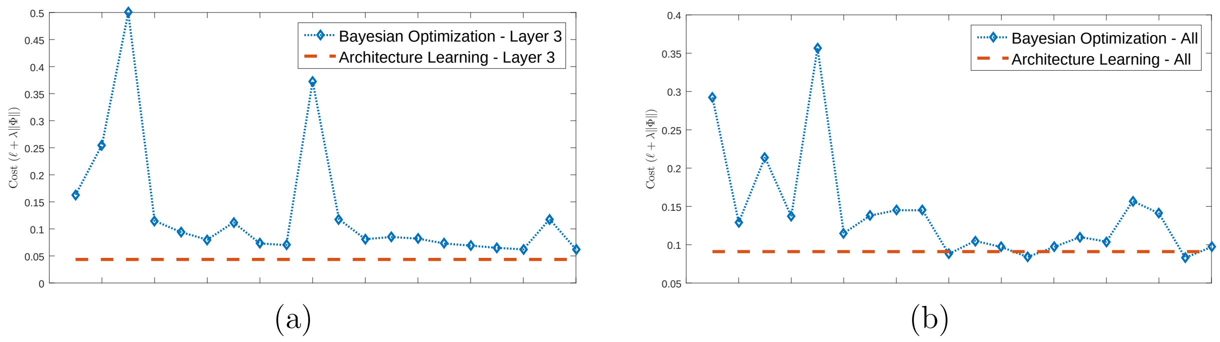


Figure 3.3: **(a)** Comparison against Bayesian Optimization for the case of learning the width of only third layer. **(b)** Similar comparison for learning the widths of all three layers. ($\lambda = 10^{-5}$ in both cases)

3.5.4 Case study: AlexNet

For the experiments that follow, we use an AlexNet-like [29] 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 [41] validation set to compute accuracies in the Table 3.4. Unlike the experiments performed 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 [33]. 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 [18] or custom implementations for diagonal matrix multiplication [54], etc.

3.5.5 A Note on Hyper-parameter selection

For effective usage of our method, we need a good set of λs. Here, we describe how to do so practically.

Method	Params	Accuracy (%)	Compression (%)
Reference Model (CaffeNet)	60.9M	57.41	0
Neuron Pruning ([45])	39.6M	55.60	35
SVD-quarter-F ([54])	25.6M	56.19	58
Adaptive FastFood 16 ([54])	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 3.4: 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 3.5: Architectures learnt by our method whose performance is given in Table 3.4.

First, we set λ_3 to a low value based on the initial widths and loss values. Recall that this value multiplies with the number of neurons in the cost function. That is, if a network has a layer with n neurons, we get $\lambda_3 \times n$. Hence, if n multiplies by 10, λ_3 divides by 10. We used 10^{-5} for MNIST-network and 10^{-6} for AlexNet. For a given initial architecture, a large λ_3 places more emphasis on getting small models than reducing loss.

Second, we set λ_1 to be about ~ 2 times λ_3. Using a positive λ_3 shifts the curve to the right. By letting $\lambda_3 = \lambda_1$, the curve shifts to the extreme right with the peak at $x = 1$. Hence if $\lambda_3 = k \times \lambda_1$, we set $0 < k < 1$.

We simply set λ_2 and λ_4 to $1/10^{th}$ of λ_1 and λ_3 respectively.

3.6 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.

Chapter 4

Connection between Architecture Learning and Dropout

4.1 Introduction

In the previous chapter, we noticed that the architecture learning objective could be written down as a regularizer. The immediate question that follows from this is - how is architecture learning related to other regularizers in literature? In this chapter, we explore connections with dropout.

Strong regularizers are often required to restrict the complexity of large deep models. Dropout [50] is a stochastic regularizer that has been widely used in recent times. However, the rule itself was proposed as a heuristic - with the objective of reducing co-adaption among neurons. As a result, it's behaviour was (and still is) not well understood. Gal and Gharamani [15] showed that dropout implicitly performs approximate Bayesian inference - making it a Bayesian Neural Net.

Bayesian Neural Nets (BNNs) view parameters of a Neural Network as random variables rather than fixed unknown quantities. As a result, there exists a distribution of possible values that each parameter can take. By placing an appropriate prior over these random variables, it is possible to restrict the model's capacity and implicitly perform regularization. The theoretical attractiveness of these methods is that one can now use tools from probability theory to work with these models. What advantages do BNNs offer over plain Neural Nets? First, they inherently capture uncertainty - both in the model parameters as well as predictions. Second, they are ideal for learning from small amounts of data. Third, a Bayesian approach has the advantage of distilling complex assumptions about the model in the form of prior distributions.

Inference over BNNs is typically intractable. As a result, one often uses approximations to the posterior distribution. MCMC and Variational Inference (VI) [6] are two popular methods for performing these approximations. In recent times, VI has emerged as the preferred method of performing this approximation as it is scalable to large models. When using VI, it is common to assume independence of model parameters. For Neural Networks, this assumption may seem unnecessarily stringent. After all, weights in a particular filter are highly correlated to produce specific patterns (an oriented edge, for instance). However, different filters in a CNN are more-or-less independent as they compute different features. In fact, it might even be advantageous to enforce independence of different filters through VI, as they reduce *co-adaptation* among features. In this work, we strive to enforce independence among features rather than weights.

The overall contributions of the chapter are as follows:

- We derive a Bayesian approach to performing inference with neural networks. In doing so, we introduce a rich family of regularizers - Generalized Dropout (GD).
- We perform experimental analysis with Dropout++, a set of methods under GD, to understand it's behaviour.
- We perform experiments with Stochastic Architecture Learning, another set of methods under GD, and show that they can be used to select the width of neural networks.
- We test Dropout++ on standard networks and show that it can be used to boost performance.

4.2 Bayesian Neural Networks

In this section, we shall formally introduce the notion of BNNs and also discuss our proposed method. Let $f(\cdot;\mathbf{w})$ denote a neural network function with parameters $\mathbf{w}$. For a given input x, the neural network produces $y = f(x;\mathbf{w})$ a probability distribution over possible labels (through softmax) for a classification problem. Given training data $\mathbf{D}$, the parameter vector $\mathbf{w}$ is updated using Bayes' Rule.

$$P(\mathbf{w}|\mathbf{D}) \propto P(\mathbf{D}|\mathbf{w})P(\mathbf{w}) \tag{4.1}$$

After computing the posterior distribution, we perform inference on a new data point x as follows. Since the neural network produces a probability distribution over labels, $P(y|x,\mathbf{w}) = f(x;\mathbf{w})$

$$P(y|x,) = \int_w P(y|x,\mathbf{w})P(\mathbf{w}|)d\mathbf{w} = \int_w f(x;\mathbf{w})P(\mathbf{w}|)d\mathbf{w} \tag{4.2}$$

Computing the posterior from equation 4.1 is intractable due to the complicated neural network structure. In Variational Inference, we define another distribution $q(\mathbf{w})$, called the variational distribution, which approximates the posterior $P(\mathbf{w}|\mathbf{D})$. This distribution is used instead of the posterior in equation 4.2 for inference.

One common assumption when working with the variational distribution is the mean-field assumption, which requires that

$$q(\mathbf{w}) = \prod_i q(w_i) \tag{4.3}$$

This states that all parameters in $\mathbf{w}$ are independent. Such an assumption is certainly not true, especially at the feature-level, where parameters are highly correlated. Works like those of Denil *et al*[12] explicitly show that parameters of a NN can be predicted given other parameters - hinting at the large amount of correlation present. Trying to enforce independence among such weights may end up having adverse effects.

It is very difficult to overcome the independence assumption within the framework of VI. In the next section, we introduce an approach to overcome this difficulty.

4.2.1 Bayesian Neural Networks with Gates

We add multiplicative gates after each feature / neuron in the neural network, as in Figure 4.1a. These gates modulate the output of each neuron. Let these new parameters be denoted by $\boldsymbol{\theta}$. Let us also assume that they lie in $[0, 1]$. Intuitively, these gates now control the *relative importance* of each particular feature as viewed by the next layer. Such relative importance may be fundamentally uncertain given a particular feature. Hence, it may be useful to think of gate parameters $\boldsymbol{\theta}$ as random variables. We shall now crystallize all these assumptions in the form of choices for the variational distribution.

We first place the following prior-hyperprior pair over the gate parameters $\boldsymbol{\theta}$, and a prior over the regular parameters $\mathbf{w}$.

$$\boldsymbol{\theta} \sim \prod_i bernoulli(k_i) \quad \mathbf{k} \sim \prod_i beta(\alpha, \beta) \quad \mathbf{w} \sim \prod_i \mathcal{N}(0, \sigma)$$

Note that the products are over all possible variables defined in the network. Here, $\mathbf{k}$ denotes the bernoulli parameters and also needs to be estimated along with $\boldsymbol{\theta}, \mathbf{w}$. Now, given that we use variational inference, let us now define the forms of the variational distributions $q(\mathbf{w}, \boldsymbol{\theta}, \mathbf{k}; \boldsymbol{\mu})$ we use. Let $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2\}$.

$$\begin{aligned} q(\mathbf{w},\boldsymbol{\theta},\mathbf{k};\boldsymbol{\mu}) &= q_1(\mathbf{w};\boldsymbol{\mu}_1)\ q_2(\mathbf{k};\boldsymbol{\mu}_2)\ q_3(\boldsymbol{\theta};\boldsymbol{\mu}_2) \\ q_1(\mathbf{w}) &= \prod_i \delta(w_i\ ;\ \mu_1^i) \qquad (4.4) \\ q_2(\mathbf{k}) &= \prod_i \delta(k_i\ ;\ \mu_2^i) \\ q_3(\boldsymbol{\theta}) &= \prod_i \mathit{bernoulli}(\theta_i\ ;\ \mu_2^i) \qquad (4.5) \end{aligned}$$

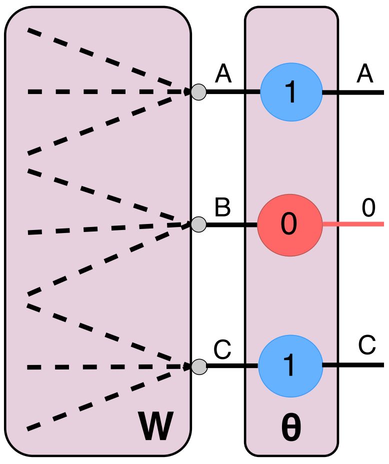
Note that even though we make an independence assumption on the weights $\mathbf{w}$ (equation 4.4), we overcome the disadvantages described in the previous section by effectively **not** being Bayesian with respect to $\mathbf{w}$, using a delta distribution. Also note that we use the same parameter $\boldsymbol{\mu}_2$ for both distributions $q_2(.)$ and $q_3(.)$. While it is true that using different parameters for both distributions could make the formulation more powerful, we use the same parameter for simplicity. Now we write equations describing the variational approximation by using the definitions above.

$$\begin{aligned} \boldsymbol{\mu}_1^*, \boldsymbol{\mu}_2^* &= \underset{\boldsymbol{\mu}_1,\boldsymbol{\mu}_2}{\arg\min} \quad \mathrm{KL}[q(\mathbf{w},\boldsymbol{\theta},\mathbf{k};\boldsymbol{\mu})||P(\mathbf{w},\boldsymbol{\theta},\mathbf{k}|\mathbf{D})] \qquad (4.6) \\ &= \underset{\boldsymbol{\mu}_1,\boldsymbol{\mu}_2}{\arg\min} \quad -\sum_\theta \log P(\mathbf{D}|\boldsymbol{\theta},\mathbf{w},\mathbf{k})\ q_3(\boldsymbol{\theta};\boldsymbol{\mu}_2) - \log P(\mathbf{k};\alpha,\beta) \end{aligned}$$

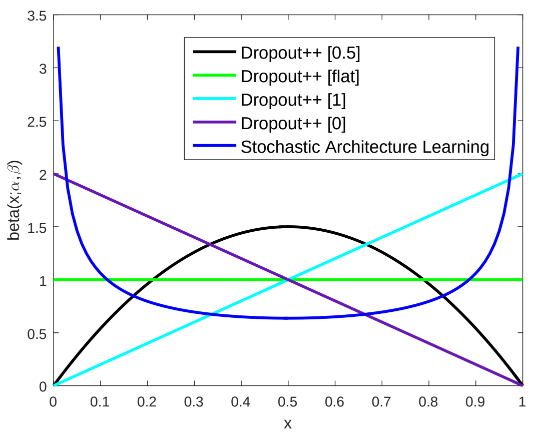
Our objective is now to solve equation 4.6. We observe that the exhaustive summation in equation 4.6 in intractable for large models. A popular method to deal with this is to use a Monte-Carlo approximation of the summation. However, even this may be infeasible for large models. As a result, we further approximate this with a *single* Monte-Carlo sample. In other words, we perform the following approximation:

$$\begin{aligned} \sum_\theta \log P(\mathbf{D}|\boldsymbol{\theta},\mathbf{w},\mathbf{k})\ q_3(\boldsymbol{\theta};\boldsymbol{\mu}_2) &\approx \log P(\mathbf{D}|\boldsymbol{\theta}^s,\mathbf{w},\mathbf{k}) \\ \text{where}\ \ \boldsymbol{\theta}^s &\sim q_3(\boldsymbol{\theta};\boldsymbol{\mu}_2) \end{aligned}$$

While this approximation seems to be drastic, we soon shall see that Classic Dropout also implicitly performs the same approximation.



(a) Neural Network Layer with Gates



(b) Beta distribution

Figure 4.1: **(a)** An illustration of the proposed method with binary stochastic multiplicative gates. Here, W refers to weights and $\boldsymbol{\theta}$ refers to gates. Note that $P(\boldsymbol{\theta} = 1) = \mathbf{k}$. **(b)** Behaviour of the beta distribution at different values of α, β.

4.2.2 Generalized Dropout

Given all the assumptions and approximations discussed above, we now write the complete objective function we aim to solve. Since the variational distributions for $\mathbf{w}$ and $\mathbf{k}$ are delta distributions, we shall now use $\mathbf{w}, \mathbf{k}$ instead of $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2$ in our notations, for simplicity.

$$\begin{aligned} \mathbf{w}^*, \mathbf{k}^* = \quad & \arg\min_{\mathbf{w},\mathbf{k}} \quad -\log P(\mathbf{D}|\boldsymbol{\theta}^s, \mathbf{w}, \mathbf{k}) - (\alpha - 1)\log \mathbf{k} - (\beta - 1)\log(1 - \mathbf{k}) \qquad (4.7) \\ & \text{where} \qquad \boldsymbol{\theta}^s \sim \mathit{bernoulli}(\boldsymbol{\theta}; \mathbf{k}) \end{aligned}$$

In the expression above, we have used the fact that $P(\mathbf{k}; \alpha, \beta)$ is a beta distribution. This form of the objective function 4.7, with gates $\boldsymbol{\theta}, \mathbf{k}$ constitutes the *Generalized Dropout* regularizer.

Let us now briefly look at the behaviour of the beta distribution at various values of α, β, as shown in Figure 4.1b. We shall refer to each of these specific cases as different versions of *Dropout++*. For reasons to be discussed later, we shall refer to the last case as *Stochastic Architecture Learning* (SAL).

- **Dropout++ (0.5)**, where $\alpha = \beta > 1$: $\mathbf{k} = 0.5$ is the most probable value of $\mathbf{k}$.
- **Dropout++ (flat)**, where $\alpha = \beta = 1$: All values of $\mathbf{k}$ are equally probable.
- **Dropout++ (1)**, where $\alpha = 1, \beta > 1$: $\mathbf{k} = 1$ is the most probable value of $\mathbf{k}$.

- **Dropout++ (0)**, where $\alpha > 1, \beta = 1$: $\mathbf{k} = 0$ is the most probable value of $\mathbf{k}$.

- **SAL**, where $\alpha < 1, \beta < 1$: $\mathbf{k} = 0$ and $\mathbf{k} = 1$ are the most probable values of $\mathbf{k}$.

Note that Dropout++ (0.5) becomes indistinguishable from Classic Dropout (with 0.5 Dropout rate) at $\alpha = \beta \to \infty$. To obtain other Dropout rates, we simply ensure $\alpha \neq \beta$. In the next section, we shall discuss another algorithm called *Architecture Learning*, and how it relates to the *SAL* method above.

4.2.3 Architecture Learning

Srinivas and Babu [46] recently introduced a method to learn the width and depth of neural network architectures. They also add additional learnable parameters similar to gates. Also, their objective function has the following form in our notation.

$$\begin{aligned} \mathbf{w}^*, \mathbf{k}^* = \quad & \arg\min_{\mathbf{w},\mathbf{k}} \quad -\log P(\mathbf{D}|\boldsymbol{\theta}^s, \mathbf{w}, \mathbf{k}) + \lambda_1 \mathbf{k}(1 - \mathbf{k}) + \lambda_3 \mathbf{k} \\ & \text{where} \qquad \boldsymbol{\theta}^s = heaviside(\mathbf{k} - 0.5) \end{aligned} \tag{4.8}$$

Note that our objective function 4.7 looks very similar to this when $\alpha, \beta < 1$ and $\alpha < \beta$, except that we use $\log \mathbf{k}$ instead of $\mathbf{k}$. Another difference is that they use a heaviside threshold to select $\boldsymbol{\theta}$ rather than sampling from a bernoulli. We observe that this is equivalent to taking a maximum likelihood sample from the bernoulli distribution. Given these similarities, we found it apt to name the corresponding method with $\alpha, \beta < 1$ as *Stochastic Architecture Learning*, as it is a stochastic version of the algorithm described above.

Most surprisingly, we find that the motivation to arrive at this algorithm was completely different - they intended to minimize the number of neurons in the network. We arrive at a very similar formulation from a purely Bayesian perspective.

4.2.4 A Practitioner's Perspective

In this section, we shall attempt to provide an intuitive explanation for Generalized Dropout. Going back to Fig. 4.1a, each neuron is augmented with a gate which learns values between 0 and 1. This is enforced by our regularizers and well as by parameter clipping. During the forward pass, we treat each of these gate values as probabilities and toss a coin with that probability. The output of the coin toss is used to block / allow neuron outputs. As a result of the learning, important features tend to have higher probability values than unimportant features.

At test time, we do not perform any sampling. Rather, we simply use the real-valued probability values in the gate variables. This approximation - called re-scaling - is used in classical Dropout as well.

What do the different Generalized Dropout methods do? Intuitively, they place restriction on the gate values (probabilities) that can be learnt. As an example, Dropout++ (0) encourages most gate values to be close to 0, with only a few important ones being high. On the other hand, Dropout++ (1) encourages gates values to be close to 1. Intuitively, this means that Dropout++ (0) restricts the capacity of a layer by a large amount, whereas Dropout++ (1) hardly changes anything. SAL, on the other hand, encourages neurons to be close to either 0 or 1. In contrast to other methods, SAL produces neural network layers that are very close to being deterministic - neurons close to 0 are almost never 'on' and those close to 1 are almost always 'on'. Dropout++ (flat) is also unique in the sense that it doesn't place any restriction on the gate values. As a result, we do not require to set any hyper-parameters for this method. From a Bayesian Perspective, when we have no prior beliefs on what the gate values should be, we use the most non-informative prior - which is Dropout++ (flat) in this case.

Dropout++ (0.5) encourages values to be close to 0.5. If the regularization constants are increased, then gate values other than 0.5 are penalized more and more heavily. In the limiting case we get Dropout, where any deviation from probability value of 0.5 is "infinitely" penalized.

4.2.5 Estimating gradients for binary stochastic gates

Given our formalism of stochastic gate variables, it is unclear how one might compute error gradients through them. Bengio *et al.* [4] investigated this problem for binary stochastic neurons and empirically verified the efficacy of different solutions. They conclude that the simplest way of computing gradients - the *straight-through* estimator works best overall. This involves simply back-propagating through a stochastic neuron as if it were an identity function. If the sampling step is given by $\theta \sim bernoulli(k)$, then the gradient $\frac{d\theta}{dk} = 1$ is used.

Another issue of consideration is that of ensuring that k always lies in $[0, 1]$ so that it is a valid bernoulli parameter. Bengio *et al.* [4] use a sigmoid activation over k. Our experiments showed clipping functions worked better. This can be thought of as a 'linearized' sigmoid. The clipping function is given by the following expression.

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

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

4.2.6 Applying to Convolution Layers

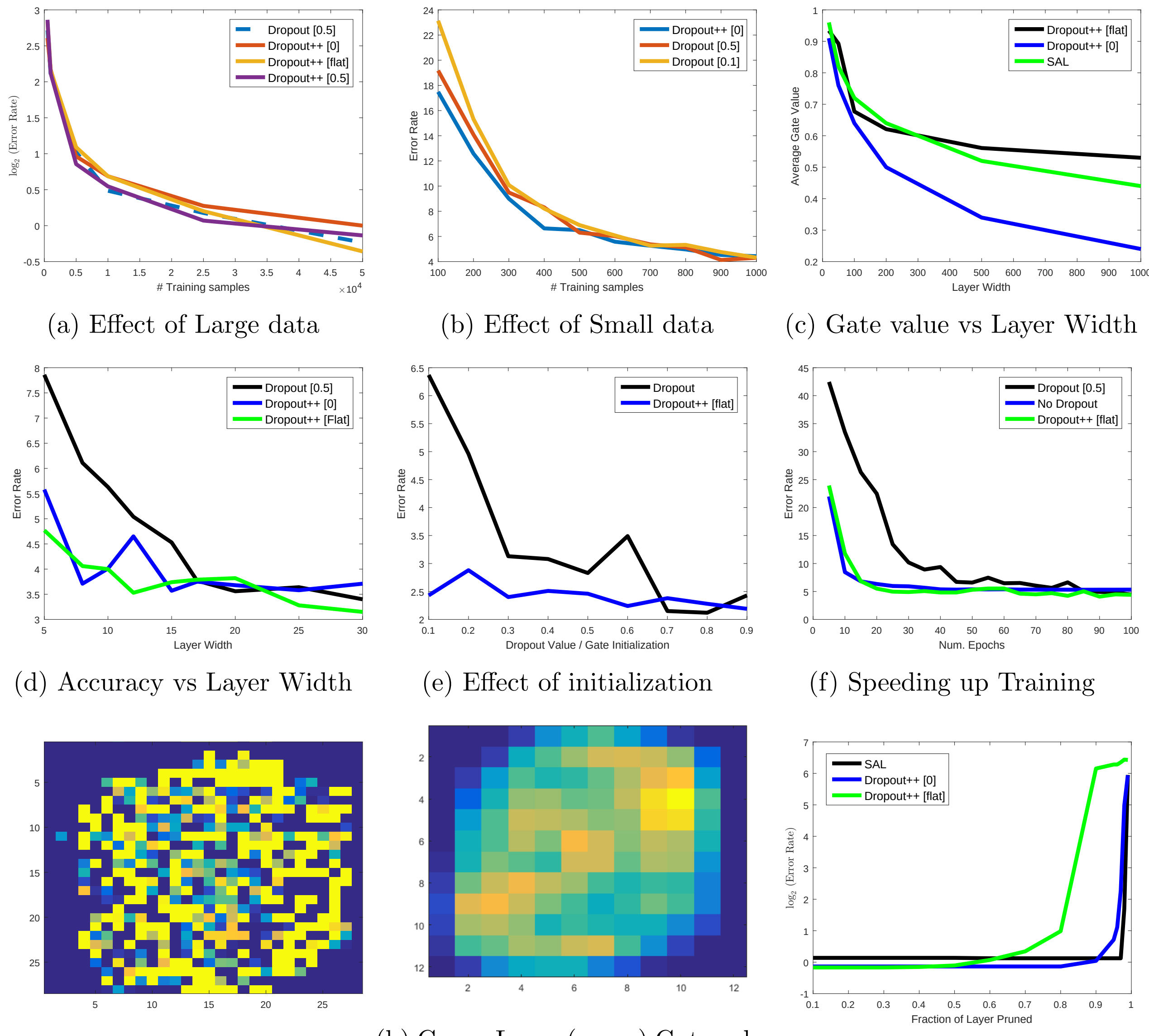
Here we shall discuss how to apply this to convolutional layers. Let us assume that the output feature map from a convolutional layer is $k \times k \times n$, i.e; n feature maps of size $k \times k$. Classical dropout samples $k \times k \times n$ bernoulli random variables and performs pointwise multiplication with the output feature map. We follow the same for Generalized Dropout as well.

However, if we wish to perform architecture selection like Architecture Learning [46], we need to select a subset of the n feature maps. In this case, we only have n gate variables, multiplying to the output of each feature map. When a gate is close to zero, and entire feature map's output becomes close to zero at test time. By selecting few feature maps out of n, we determine which of the n filters in the previous layer are essential.

4.3 Related Work

There are plenty of works which aim to extend Dropout. DropConnect [53] stochastically drops weights instead of neurons to obtain better accuracy on ensembles of networks. As stated earlier, using the independence assumption for weights may not be correct. Indeed, DropConnect is shown to work on only fully connected layers. Standout [3] is a version of Dropout where the dropout rate depends on the output activations of a layer. Variational Dropout [28] proposes a Bayesian interpretation for Gaussian Dropout rather than the canonical multiplicative Dropout. By considering multiplicative Dropout, we make important connections to Architecture Learning / neuron pruning. Gal and Gharamani [15] showed a Bayesian interpretation for binary dropout and show that test performance improves by performing Monte-Carlo averaging rather than re-scaling. For simplicity, we use the re-scaling method at test time for Generalized Dropout. Our work can be seen as an extension of this work by considering a hyper-prior along with a bernoulli prior.

Hinton and Van Camp [23] first introduced variational inference for making Neural Networks Bayesian. Recent work by Graves [17] and Blundell et al. [7] further investigated this notion by using different priors and relevant approximations for large networks. Probabalistic Backpropagation [22] is an algorithm for inferring marginal posterior probabilities for special classes of Bayesian Neural Networks. Our method is different from any of these methods as they are all Bayesian over the weights, whereas we are only Bayesian with respect to the gates.



(a) Effect of Large data (b) Effect of Small data (c) Gate value vs Layer Width

(d) Accuracy vs Layer Width (e) Effect of initialization (f) Speeding up Training

(g) Input Layer Gate values (h) Conv. Layer (mean) Gate values (i) Pruning Gates

Figure 4.2: **(a, b)** All Dropout++ methods perform on par with Dropout on large data, and Dropout++ (0) seems to work better on small data. **(c, d)** Dropout++ can adapt to different layer sizes and result in optimal performance. **(e, f)** Initialization of Dropout++ parameters is not very crucial. As a result, one can initialize favourably to drastically increase training speed. **(g, h)** Dropout++ on convolutional layers learns to selectively attend to parts of the image, rather than the full image. **(i)** SAL and Dropout++ (0) are best suited for gate pruning, with SAL being better for automated pruning given the steep slope.

4.4 Experiments

In this section, we perform experiments with the Generalized Dropout family to test their usefulness. First, we perform a wide variety of analysis with the Generalized Dropout family.

Method	Architecture	Error (%)	No. of Params
Baseline	20-50-500-10	0.82	431k
Architecture Learning [46]	20-50-20-10	0.93	41.8k
SAL [$\beta/\alpha = 1$]	18-50-296-10	0.69	263k
SAL [$\beta/\alpha = 10$]	11-33-38-10	0.84	**29.8k**
SAL [$\beta/\alpha = 100$]	7-13-16-10	1.14	5.9k

Table 4.1: Architecture selection capability of our method on a LeNet-like baseline. The first row is the architecture and performance of the baseline LeNet network. We see that larger β/α ratios result in smaller networks at the cost of network performance.

Later, we study some specific applications of this method. We perform experiments primarily using Theano [5] and Lasagne.

4.4.1 Analysis of Generalized Dropout

We shall now analyze the behaviours of different members of Generalized Dropout family to find out which ones are useful. For the experiments on the MNIST dataset, we use the standard LeNet-like architecture [32], which consists of two 5×5 convolutional layers with 20 and 50 filters, and two fully connected layers with 500 and 10 (output layer) neurons. While there is nothing particularly special about this architecture, we simply use this as a standard net to analyze our method.

4.4.1.1 Effect of data-size

We investigate whether Generalized Dropout indeed has any advantage over Dropout in terms of accuracy. Here, we apply Dropout and Generalized Dropout only to the last fully connected layer. Our experiments reveal that for the network considered, the accuracies achieved by any Generalized Dropout method are not always strictly better than Dropout, as shown in Figure 4.2a. This indicates that most of the *regularization power* of Dropout comes from the independence assumption of Variational Inference, rather than particular values of the dropout parameter. This is a surprising result which we shall use to our advantage in the paper.

However, we note that for small data-sizes, Dropout++ (0) seems to be advantageous over Dropout (Figure 4.2b). This is possibly because Dropout++ (0) forces most neurons (but not all) to have very low capacity due to low value of the parameters. [1]

4.4.1.2 Effect of Layer-width

Inspired from the above results about Dropout++ (0), we look at the relationship between using different layer-widths for the fully connected layer and the learnt gate parameters. Intuitively,

[1]Note that in our notation, a large value of Dropout++ indicates a large probability of retaining the neuron, contrary to popularly used notation for Dropout.

Model	**Original**	**D++** (1.0)	**Drop** (0.98)	**D++** (0.5)	**Drop** (0.8)
ResNet-32 [21]	7.46	**6.73**	6.8	-	-
ResNet-56 [21]	6.75	**6.1**	6.18	-	-
GenericNet [44]	6.86	**6.6**	6.75	**6.32**	6.46

Table 4.2: Applying Dropout++ after each layer in standard networks decreases error rate (%). Dropout++ learnt values (with init = 1.0) are close to 0.98. As a result, Dropout at $p = 0.98$ performs similar to this learnt value. With Dropout++ init = 0.5, the learnt values are close to $p = 0.8$.

it is natural to assume that larger layers should learn lower gate values, whereas smaller layers should learn much higher values, if we wish for the overall *capacity* of the layer to remain roughly the same. Our experiments confirm this intuition as shown in Figure 4.2c.

We also test if this flexibility translates to higher accuracy numbers over a fixed dropout value, and we find this to be indeed the case. We find that for small layer-widths, Dropout (at $p = 0.5$ for example), tends to remove too many neurons, while Dropout++ adjusts it's parameter values to account for small layer-widths, as shown in Figure 4.2d.

4.4.1.3 Effect of Initialization

Initialization of good parameters is known to play a key-role in generalization of deep learning systems. To test whether this holds for the newly introduced Generalized Dropout parameters as well, we try different initializations of the Generalized Dropout parameters. In this example, we simply initialize all gates to a single constant value. As expected, we find that the choice of this initialization is much less crucial when compared to setting the Dropout value, as shown in Figure 4.2e.

The choice of initialization, however, affects training time. As an example, it is empirically observed that Dropout with $p = 0.1$ is much slower than $p = 0.9$. Therefore, it is helpful to have higher Dropout rates to facilitate faster training. To help faster training in Dropout++, we simply initialize with $p = 1.0$, i.e; start with a network with no Dropout and gradually learn how much Dropout to add. We observe that this indeed helps training time and at the same time provides the flexibility of Dropout, as shown in Figure 4.2f.

4.4.1.4 Visualization of Learnt Parameters

Until this point, we have focussed on using Generalized Dropout on the fully connected layers. Similar effects hold when we apply these to convolutional layers as well. Here, we visualize the learnt parameters in convolutional layers. First, we add Dropout++ only to the input layer. The resulting gate parameters are shown in Figure 4.2g. We observe a similar effect when we add Dropout++ only to the first convolutional layer, as shown in Figure 4.2h, which shows the average gate map of all the convolutional filters in that layer. In both cases, we observe that Dropout++ learns to selectively attend to the centre of the image rather than towards the corners.

This has multiple advantages. First, by not looking at the corners of each feature, we can potentially decrease model evaluation time. Second, this breaks translation equivariance implicit in convolutions, as in our case certain spatial locations are more important for a filter than others. This could be helpful when using CNNs for face images (for example), where a filter need not look for an "eye" everywhere in the image. Such locally connected layers have been previously used in works such as DeepFace [52]. Dropout++ could offer a more natural way to incorporate such an assumption.

4.4.1.5 Architecture Selection

We shall now attempt to use Stochastic Architecture Learning (SAL) to automatically learn the required layer width of the network. The inherent assumption here is that the initial architecture is over-complete, and that a sub-set of neurons is sufficient to get similar performance. We first learn the parameters of the network using SAL regularizer, later we prune neurons with low gates parameters. Figure 4.2i shows that SAL learns gate parameters that are often close to either 0 or 1, resulting in a much sharper rise compared to the other methods. We use this sharp rise as a criterion to select the width of a layer. We observe that varying the β/α parameter encourages the method to get smaller architectures, sometimes at the cost of accuracy, as shown in Table 4.1.

4.4.2 Dropout++ on standard models

So far we have studied the various properties of Generalized Dropout by performing various experiments on LeNet. We shall now shift to larger networks to test the effectiveness of Dropout++. Modern networks mainly use dropout only in the fully connected layers, or simply not at all, owing to much powerful regularizers such as Batch Normalization. Here we shall take such networks, simply add Dropout++ (flat) after each layer, and see if we get an increase in accuracy. We perform experiments with ResNet32, ResNet56 and a Generic VGG-like network, all trained on the CIFAR-10 dataset. As in Table 4.2, we see that for all three models, adding Dropout++ is largely helpful.

4.5 Conclusion

We have proposed *Generalized Dropout*, a family of methods that generalize Dropout-like behaviour. One set of methods in this family, *Dropout++*, is an adaptive version of Dropout. *Stochastic Architecture Learning* is another set of methods that performs architecture selection. An uninformed choice of the Dropout parameter usually hurts performance. Dropout++ helps in setting a useful parameter value regardless of factors such as layer width and initialization. Experiments show that it is generally beneficial to simply add Dropout++ (flat) after every layer of a Deep Network.

Chapter 5

Application: Training Sparse Neural Networks

5.1 Introduction

In this chapter we shall discuss the application of techniques discussed previously to training sparse neural networks. While the methods of previous chapters introduced block sparsity in weights, here we impose full sparsity. In doing so, we also briefly discuss connections of the general method to spike-and-slab priors and stochastic optimization.

Large networks with many millions of parameters [29], [42], [51] often use dense matrix multiplications and convolutions. However, these networks typically use dense computations. Would it be advantageous to use sparse computations instead? 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]. Further, having a lower parameter count may help in avoiding overfitting.

Regularizers are often used to discourage overfitting. 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 in inducing sparsity in case of large non-convex problems like deep neural network training as shown in [11]. The contribution of this chapter is to be able to induce sparsity in a tractable way for such models.

The overall contributions of the chapter are as follows.

[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

- We propose a novel regularizer that restricts the total number of parameters in the network.
- We perform experimental analysis to understand the behaviour of our method.
- We apply our method on LeNet-5, AlexNet and VGG-16 network architectures to achieve state-of-the-art results on network compression.

5.2 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 2 *The complexity of a m-layer neural network with index set Φ is given by* $\|\Phi\| = \sum\limits_{i=1}^{m} n_i$.

We now aim to solve the following optimization problem.

$$\hat{\theta}, \hat{\Phi} = \underset{\theta, \Phi}{\arg\min}\ \ell(\hat{y}(\theta, \Phi), y) + \lambda \|\Phi\| \tag{5.1}$$

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 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 $\in \{0,1\}$.

5.2.1 Gate Variables

How do we incorporate the index set formalism in neural networks? Assume that the index set (G^s in Fig. 5.1) is multiplied pointwise with the weight matrix. This results in a weight matrix

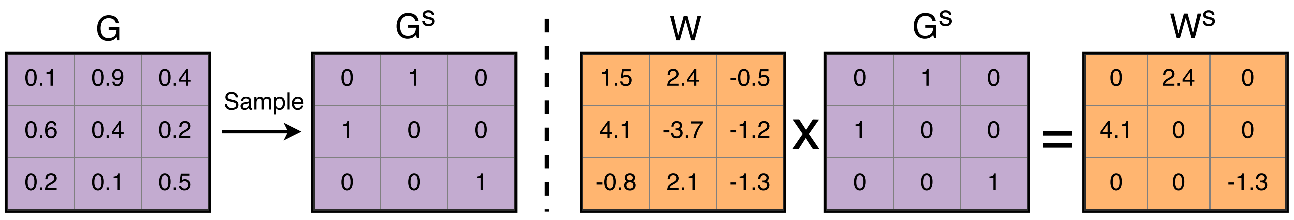


Figure 5.1: Our strategy for sparsifying weight matrices. First, we sample / threshold the gate variables. We then multiply the resulting binary matrix with W, to yield a sparse matrix W^s.

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. 5.1), 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.

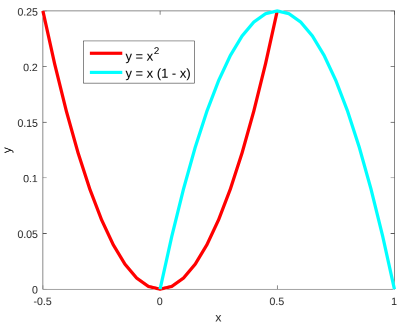
When we draw from a bernoulli distribution, we have two choices - we can either perform a *unbiased* draw (the usual sampling process), or we can perform a so-called *maximum-likelihood (ML)* draw. The ML draw involves simply thresholding the values of G at 0.5. To ensure determinism, we use the ML draw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

5.2.2 Promoting Sparsity

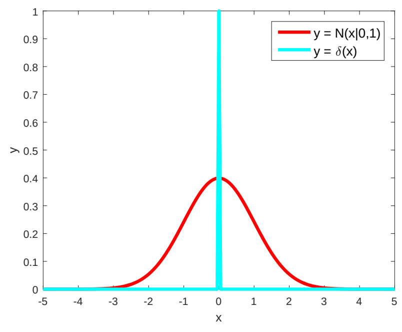
Given our formalism of gate variables, how do we ensure that the learnt bernoulli parameters are low - or in our case - mostly less than 0.5 ? One plausible option is to use the ℓ_2 or the ℓ_1 regularizer on the gate variables. However, this does not ensure that there will exist values greater than 0.5. 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.

To this end, we use a regularizer given by $w \times (1 - w)$. This was introduced by [36] 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 Fig. 5.2a

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



(a) Bi-modal regularizer



(b) Spike and slab prior

Figure 5.2: **(a)** The bi-modal regularizer used in our work. Note that this encourages values to be close to 0 and 1, in contrast to the ℓ_2 regularizer. **(b)** An example of a spike-and-slab prior similar to the one used in this work (except for a constant).

stated as follows.

$$\hat{\theta}, \hat{\Phi} = \underset{\theta,\Phi}{\arg\min}\; \ell(\hat{y}(\theta, \Phi), y) + \lambda_1 \sum_{i=1}^{m} \sum_{j=1}^{n_i} g_{i,j}(1 - g_{i,j}) + \lambda_2 \sum_{i=1}^{m} \sum_{j=1}^{n_i} g_{i,j} \tag{5.2}$$

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. 5.2 vanishes and the third term becomes $\lambda\|\Phi\|$, thus reducing to Eqn.5.1.

5.2.3 An Alternate Interpretation

Now that we have arrived at the objective function in Eqn.5.2, it is natural to ask the question - how do we know that it solves the original objective in Eqn.5.1 ? We shall now derive Eqn.5.2 from this perspective.

Assuming the formulation of gate variables, we can re-write the objective in Eqn.5.1 as follows.

$$\begin{aligned} \hat{\theta}, \hat{\Phi} &= \underset{\theta,G}{\arg\min}\; \ell(\hat{y}(\theta, G^s), y) + \lambda \sum_{i=1}^{m} \sum_{j=1}^{n_i} g^s_{i,j} \\ g^s_{i,j} &\sim \text{bernoulli}(g_{i,j}),\ \forall\ i, j \end{aligned} \tag{5.3}$$

where g^s is the sampled version of gate variables g. Note that Eqn.5.3 is a stochastic objective function, arising from 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.

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

where $\mathbf{E}(g_{i,j}^s) = g_{i,j}$. 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.5.2.

5.2.4 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 [34]. 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.

$$
P(W) = \frac{1}{Z} \prod_i \ exp(- \ (1 - \delta(w_i)))^{\alpha} \ \mathcal{N}(w_i | 0, \sigma^2)^{1-\alpha} \tag{5.4}
$$

Here, $\delta(\cdot)$ denotes the dirac delta distribution, and Z denotes the normalizing constant, and α is the mixture coefficient. Also note that like [34], we assume that $w_i \in [-k, k]$ for some $k > 0$. This is visualized in Fig. 5.2b. 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

$$
-\log P(W) = -\alpha \sum_i (1 - \delta(w_i)) \ + \ \frac{1-\alpha}{2\sigma^2} \sum_i w_i^2 \tag{5.5}
$$

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. 3.1. 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. 5.4 is a spike-and-slab prior which we implicitly end up using in this method.

5.2.5 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.* [4] 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 identity function. If the sampling step 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.* [4] 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.

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

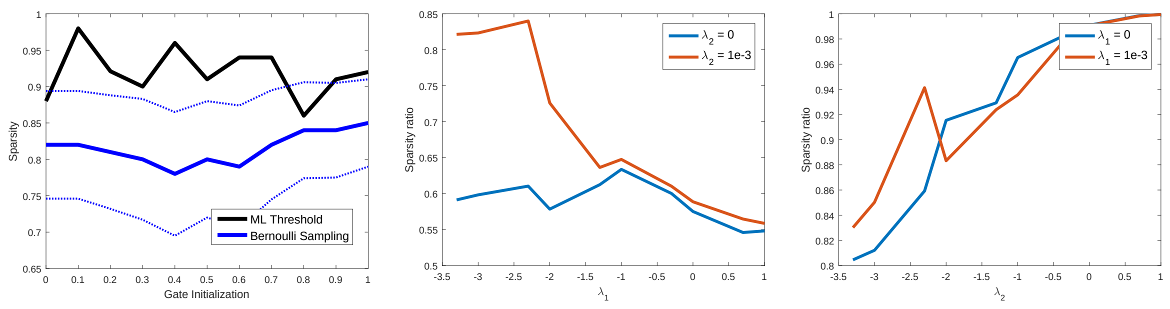
The overall sampling function is hence given by $g^s \sim bernoulli(clip(g))$, and the straight-through estimator is used to estimate gradients overall.

5.2.6 Comparison with LASSO

LASSO is 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.

5.2.7 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 the two to result in sparse matrices which considerably reduces model size at test time.



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

Figure 5.3: **(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.

Essentially we do not have to store both sets of parameters while testing,only a element-wise product of the two is required. 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 that fully connected layers due to the small number of parameters they possess.

5.3 Related Work

There have been many recent works which perform compression of neural networks. Weight-pruning techniques were popularized by LeCun *et al.* [31] and Hassibi *et al.* [19], who introduced *Optimal Brain Damage* and *Optimal Brain Surgery* respectively. Recently, Srinivas and Babu [45] proposed a neuron pruning technique, which relied on neuronal 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.* [18] create sparse networks by alternating between weight pruning and network training. A similar strategy is followed by Collins and Kohli [11]. 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.

Many methods have been proposed to train models that are deep, yet have a lower parameterisation than conventional networks. Denil *et al.* [12] 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.* [54] 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.* [38] use tensor decompositions to obtain a factorization of tensors with small number of parameters. Cheng *et al.* [9] 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.* [16] 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.

Our work is very similar to that of Architecture Learning [46], which uses a similar framework to minimize the total number of neurons in a neural network. On the other hand, we minimize the total number of weights.

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 5.1: Compression results for LeNet-5 architecture.

5.4 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 [32]. Second, we use our method to perform network compression on two networks - LeNet-5 and AlexNet. These networks are trained on MNIST and ILSVRC-2012 dataset respectively. Our implementation is based on Lasagne, a Theano-based library.

5.4.1 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

Method	**Params (P+I)***	**Accuracy(%)**	**Compression Rate(%)**
Baseline	431K	99.20	1x
SVD(rank-10)[13]	43.6K	98.47	10x
AL [46]	40.9K	99.04	10.5x
AF-1024 [54]	38.8K	99.29	11x
Han *et al*[18]	36K (72k)*	99.23	12x
Method-1	18K (36k)*	99.19	24x
Method-2	22K (44k)*	99.33	19x

Table 5.2: Comparison of compression performance on LeNet-5 architecture. *Count of number of parameters and the indices to store them. This is the effective storage requirement when using sparse computations.

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 5.3: Layer-wise compression performance on AlexNet

5.4.1.1 Effect of hyper-parameters

In Section 2.1 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 5.5, both the

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 5.4: Layer-wise compression performance on VGG-16

thresholding as well as the sparsity-based methods are similarly sensitive to the regularization constants.

In Section 2.3, we saw that λ_1 roughly controls the variance of the bernoulli variables while λ_2 penalizes the mean. In Table 5.5, we see that the mean sparsity for the pair $(\lambda_1, \lambda_2) = (1, 0)$ is high, while that for $(0, 1)$ is considerably low. 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.

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 5.3b we see that λ_1, or the variance-controlling hyper-parameter, mainly *stabilizes* the training by reducing the sparsity levels. In Figure 5.3c we see that increasing λ_2 increases the sparsity level as expected.

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

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

Method	Total Params (P + I)	Accuracy(%)	Compression Rate
Baseline	60.9M	57.2	-
Neuron Pruning [45]	39.6M	55.60	1.5x
SVD-quarter-F [54]	25.6M	56.18	2.3x
Adaptive FastFood 32 [54]	22.5M	57.39	2.7x
Adaptive FastFood 16 [54]	16.4M	57.1	3.7x
ACDC [35]	11.9M	56.73	5x
Collins & Kohli [11]	8.5M (17M)	55.60	7x
Han *et al*[18]	6.7M (13.4M)	57.2	9x
Proposed Method	**5.9M** (11.8M)	**56.96**	**10.3x**

Table 5.6: Comparison of compression performance on AlexNet architecture

Method	Total Params (P + I)	Accuracy(%)	Compression Rate
Baseline	138M	68.97	-
Han *et al*[18]	10.3M (20.6M)	68.66	13x
Proposed Method	**9.8M** (19.6M)	**69.04**	**14x**

Table 5.7: Comparison of compression performance on VGG-16 architecture

particular advantage in doing so. As shown in Figure 5.3a, 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.

5.4.2 Compression Performance

We test compression performance on three different network architectures - LeNet-5, AlexNet [29] and VGG-16.

For LeNet-5, we simply sparsify each layer. As shown in Table 5.1, we are able to remove about 96% of LeNet's parameters and only suffer a negligible loss in accuracy. Table 5.2 shows that we obtain state-of-the-art results on LeNet-5 compression. For Proposed Method - 1, we used $(\lambda_1, \lambda_2) = (0.001, 0.05)$, while for Proposed Method-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. As a result, we report the parameter count along with indices. This is similar to what has been done in [11]. 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 starting to fine-tune. 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 88% sparsity.

We run fine-tuning on AlexNet for 30k iterations ($\sim$ 18 hours), and VGG for 40k ($\sim$ 24 hours) iterations before stopping training based on the combination of compression ratio and validation accuracy. This is in contrast with [18], 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 5.6 and Table 5.7, we obtain favourable results when compared to the other network compression / sparsification methods.

5.5 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 achieve state-of-the-art results on compression of deep neural networks.

Chapter 6

Conclusion

In this thesis, we have broadly examined the problem of removing neurons from a neural network. To this end, we have developed a generic set of tools based on adding binary multiplicative gates to parameters.

In the second Chapter, we found that even a naive data-free method can obtain significant compression ratios for pruning deep networks. This exposes a fundamental issue with the way we build and design deep neural networks. Ideally, we should be able to design models that have little or no redundancy in their architectural specification. This question led us to the content of Chapter 3, where we developed a method to automatically find a small neural network architecture for that training data. We theoretically showed that the resultant neural networks do not contain redundant neurons.

Admittedly, deciding whether to either discard a given neuron or not is too harsh. In reality, there might exist graded levels of importance. In an attempt to extend Architecture Learning to incorporate such grey areas, we inadvertently discovered a connection to Dropout. This was discussed in Chapter 4. Specifically, it turned out that *Stochastic Architecture Learning* (SAL) shows the intended behaviour. By simply exploring other values of the underlying beta-regularizer, we stumbled upon Dropout++ - an adaptive family of Dropout. In a way, Dropout and Architecture Learning are opposites of one another. While Dropout temporarily drops neurons to produce good generalization, Architecture Learning permanently drops them to reduce model size. Curiously, Dropout occurs at $\alpha, \beta \to \infty$, while Architecture Learning is at $\alpha, \beta < 1$, occurring at the extreme ends of the spectrum. SAL attempts to compromise between the two behaviours - it stochastically removes neurons at the start of training, realises that some neurons are not required, and gradually removes them toward the end of optimization. Indeed, results show that SAL outperforms our previous method - enabling us to compress models better.

The tools developed in Chapters 3 and 4 are rather general. To show their generality, we apply these to the problem of removing weights in a neural network (Chapter 5). This method ended up outperforming some of the state-of-the-art methods. In order to connect with the previous work is this area, we made a connection of our method with sparsity inducing regularizers. We found that our technique is reminiscent of a class of priors called *Spike-and-slab* priors. These are alternatives to traditional ℓ_1 based regularizers, which provide sparse solutions to convex problems. However, given that neural network training is non-convex, ℓ_1-based methods do not work as well. We empirically find that our proposed method better optimizes the underlying ℓ_0 optimization problem. We hypothesize that this regularizer can be used to encourage sparsity for other large non-convex problems as well.

Bibliography

[1] Arthur Asuncion and David Newman. UCI machine learning repository, 2007. 20

[2] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662, 2014. 12

[3] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092, 2013. 45

[4] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 44, 56

[5] 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. 33, 47

[6] Christopher M Bishop. Pattern recognition. *Machine Learning*, 2006. 3, 39

[7] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015. 45

[8] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006. 6, 12, 17

[9] Yu Cheng, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu 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. 58

[10] Dan C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011. 13

[11] Maxwell D. Collins and Pushmeet Kohli. Memory bounded deep convolutional networks. *CoRR*, abs/1412.1442, 2014. URL http://arxiv.org/abs/1412.1442. 7, 13, 32, 51, 57, 61

[12] Misha Denil, Babak Shakibi, Laurent Dinh, and Nando de Freitas. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013. 13, 32, 40, 57

[13] 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. 13, 32, 59

[14] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. 1989. 32

[15] Yarin Gal and Zoubin Ghahramani. Bayesian convolutional neural networks with bernoulli approximate variational inference. *arXiv preprint arXiv:1506.02158*, 2015. 38, 45

[16] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 13, 32, 58

[17] Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011. 45

[18] 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. 7, 32, 36, 57, 59, 61, 62

[19] 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. 12, 32, 57

[20] 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. 27

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 48

[22] José Miguel Hernández-Lobato and Ryan P Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. *arXiv preprint arXiv:1502.05336*, 2015. 45

[23] Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 5–13. ACM, 1993. 45

[24] Geoffrey E Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS 2014 Deep Learning Workshop*, 2014. 6, 13, 17

[25] 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. 13, 32

[26] 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. 21

[27] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015. vii, 9, 10

[28] Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *arXiv preprint arXiv:1506.02557*, 2015. 45

[29] 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. 5, 12, 15, 20, 22, 36, 51, 61

[30] Praveen Kulkarni, Joaquin Zepeda, Frederic Jurie, Patrick Prez, and Louis Chevallier. Learning the structure of deep architectures using l1 regularization. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 23.1–23.11. BMVA Press, 2015. 32

[31] 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. 12, 32, 57

[32] 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. 2, 21, 33, 47, 58

[33] 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. 36

[34] Toby J Mitchell and John J Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83(404):1023–1032, 1988. 55

[35] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. ACDC: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*, 2015. 61

[36] Walter Murray and Kien-Ming Ng. An algorithm for nonlinear optimization problems with binary variables. *Computational Optimization and Applications*, 47(2):257–288, 2010. 28, 53

[37] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010. 5

[38] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015. 58

[39] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016. 8, 9

[40] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014. 13

[41] 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. 2, 22, 36

[42] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015. 5, 12, 20, 51

[43] 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. 32, 35

[44] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Patwary, Mostofa Ali, and Ryan P Adams. Scalable bayesian optimization using deep neural networks. *arXiv preprint arXiv:1502.05700*, 2015. 48

[45] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 31.1–31.12. BMVA Press, 2015. iii, 32, 37, 57, 61

[46] Suraj Srinivas and R Venkatesh Babu. Learning the architecture of deep neural networks. *arXiv preprint arXiv:1511.05497*, 2015. iii, 43, 45, 47, 58, 59

[47] Suraj Srinivas and R Venkatesh Babu. Generalized dropout. *arXiv preprint arXiv:1611.06791*, 2016. iii

[48] Suraj Srinivas, Ravi Kiran Sarvadevabhatla, Konda Reddy Mopuri, Nikita Prabhu, Srinivas SS Kruthiventi, and R Venkatesh Babu. A taxonomy of deep convolutional neural nets for computer vision. *Frontiers in Robotics and AI*, 2:36, 2016. iii, 2

[49] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. Training sparse neural networks. *arXiv preprint arXiv:1611.06694*, 2016. iii

[50] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 12, 38

[51] 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. 5, 12, 51

[52] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014. 4, 49

[53] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013. 45

[54] 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. 9, 32, 36, 37, 58, 59, 61

[55] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in Neural Information Processing Systems 27*, pages 487–495. 2014. 2