# DropBack: Continuous Pruning During Deep Neural Network Training

by

Maximilian Golub

B. Electrical Engineering, University of Washington, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Applied Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

September 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**DropBack: Continuous Pruning During Deep Neural Network Training**

submitted by **Maximilian Golub** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

**Examining Committee:**

Mieszko Lis, Electrical and Computer Engineering
*Supervisor*

Guy Lemieux, Electrical and Computer Engineering
*Supervisor*

Jane Wang, Electrical and Computer Engineering
*Supervisory Committee Member*

Matei Ripeanu, Electrical and Computer Engineering
*Supervisory Committee Member*

# Abstract

In recent years, neural networks have regained popularity in a variety of fields such as image recognition and speech transcription. As deep neural networks grow more popular for solving everyday tasks, deployment on small embedded devices — such as phones — is becoming increasingly popular. Moreover, many applications — such as face recognition or health applications — require personalization, which means that networks must be retrained after they have been deployed.

Because today's state-of-the-art networks are too large to fit on mobile devices and exceed mobile device power envelopes, techniques such as pruning and quantization have been developed to allow pre-trained networks to be shrunk by about an order of magnitude. However, they all assume that the network is first fully trained off-line on datacenter-class GPUs, then pruned in a post-processing step, and only then deployed to the mobile device.

In this thesis, we introduce DropBack, a technique that significantly reduces the storage and computation required during *both inference and training*. In contrast to existing pruning schemes, which retain the weights with the largest values and set the rest to zero, DropBack identifies the weights *that have changed the most*, and *recomputes* the original initialization values for all other weights. This means that only the most important weights must be stored in off-chip memory both during inference and training, reducing off-chip memory accesses (responsible for a majority of the power usage) by up to 72×.

Crucially, networks pruned using DropBack maintain high accuracy even for challenging network architectures: indeed, on modern, compact network architectures such as Densenet and WRN-28-10, DropBack outperforms the current state-of-the-art pruning techniques in both accuracy and off-chip memory storage required for weights. On the CIFAR-10 dataset, we observe 5× reduction in weights on an already 9×-reduced VGG-16 network, which we call VGG-S, and 4.5× on Densenet and WRN-28-10 — all with zero or negligible accuracy loss — or 19×, 27×, and 36×, respectively, with a minor impact on accuracy. When the recomputed initial weights are decayed to zero, the weight memory footprint of WRN-28-10 can be reduced up to 72×.

# Lay Summary

Machine learning models power many consumer-facing features such as Apple's personal assistant Siri and Tesla's self-driving cars. Because current models are too large to store and adapt on mobile devices like smartphones, any model improvements and updates must be done off-line using cloud resources. In many applications, however, transmitting data to the cloud is unacceptable for privacy and security reasons, and in such use cases the models cannot be retrained after deployment.

In this thesis, we aim to solve this issue by making it possible to train modern machine learning models on these smaller, energy-constrained systems. We develop DropBack, a method for reducing the on-device storage costs needed for creating — i.e., training — large machine learning models. Unlike previous research, which has focused on reducing the costs of *using* already created models, DropBack reduces the costs of both creating *and* using such models. Compared to prior work, DropBack decreases the on-device storage requirements by an order of magnitude, all while retaining best-in-class accuracy results on modern image recognition networks.

# Preface

This thesis is original, independent work by the author, Maximilian Golub. Professor Mieszko Lis and Professor Guy Lemieux served as advisors, and helped to edit this work.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Neural networks are becoming an everyday part of modern life; from voice assistants in cellphones to Amazon shopping recommendations, neural networks are ubiquitous. Indeed, some have likened the resurgence of neural networks to the next dotcom-level boom [5, 13, 59]. Typically, these neural networks are deployed in the cloud using datacenter-class GPU compute resources; increasingly, however, they are also being deployed on mobile devices, e.g., for voice and facial recognition [61, 62].

In comparison to top-end GPUs, mobile devices have very limited off-chip DRAM bandwidth and drastically reduced power envelopes. Because modern deep neural networks are large and computationally expensive to use — and even more computationally expensive to train — bringing the recent advances in neural network technology to mobile devices remains a challenge. Consequently, much work in the last three years has focused on reducing *pre-trained* neural networks to fit on mobile devices so that they can be used for *inference* [14, 18, 64, 68].

Comparatively little attention, however, has been paid to enabling *training* within the low power envelopes of mobile devices. On-device training is desirable to (re-)train networks to improve accuracy for individuals users' voices and faces [61, 62], as well as in other applications where transmitting training data to the cloud is undesirable for privacy and security reasons. This presents three key challenges: (a) state-of-the-art networks need to be trained within the mobile device off-chip DRAM size and off-chip bandwidth restrictions; (b) the power and energy costs of training, which include computation and off-chip memory access, must be reduced to fit within the mobile device power and energy budgets; and (c) models trained on mobile devices should not lose significant accuracy compared to models trained in the cloud.

In this thesis, we take a step towards enabling on-device training by developing DropBack, an algorithm that lowers the off-chip memory usage for weights, and optionally reduces the computation requirements, during training. We demonstrate that DropBack can reduce the number of weights that must be stored during training by up to 5× with no loss of accuracy, and up to 36× if a small amount of accuracy can be sacrificed. Because retrieving weights in off-chip DRAM and computing on it consumes substantial energy, reducing the memory footprint can in turn reduce the required power envelope; if DropBack is incorporated in a neural network accelerator — a task outside of the scope of this thesis — it can potentially reduce the energy required to train deep neural networks on mobile devices by an order of magnitude.

## 1.1 Energy Consumption of Large Neural Networks

### Energy Usage During Inference

Modern deep neural networks can use large amounts of storage and computation: for example, the WRN-28-10 network [71] requires 144MB of weights to be accessed in off-chip DRAM and 5.48 billion floating-point multiply-accumulate operations to classify a single image. In the 45nm technology node, this requires a minimum of 47mJ of energy for accessing DDR2 DRAM (2.6nJ per 64 bits accessed [25]) and a further 27mJ for computation (estimating a single floating-point multiply-accumulate at 5pJ = 0.9pJ + 4pJ per 32-bit floating-point fused multiply-accumulate [25]). In addition, a realistic chip would also have other overheads, such as scheduling, accessing on-chip register files and scratchpads, and so on. To label images at the rate of 60 images per second, WRN-28-10 would consume a minimum of 2.8W[1] of power on memory accesses and 1.6W[2] on computation.

[1] $144MB * 60s^{-1} * 2600picojoules/64b = 2.8W$

[2] $5.48 * 10^9 ops * 60s^{-1} * 5picojoules/op = 1.6W$

## Reducing Power During Inference

Because accessing DRAM costs ~650× more than a floating-point multiply [25], reducing the number of weights present in the trained network can considerably reduce the energy used for inference.

One simple method of reducing the weight count in neural networks is pruning [14, 18, 38, 44, 46, 60, 64, 65, 74]. Pruning removes some — ideally most — of the weights of a neural network, usually by dropping the weights with the lowest values (we refer to this as *magnitude-based pruning*). An example of this pruning technique is shown in Figure 1.2: a simple multi-layer perceptron with 13 weights is first trained using the full network (Figure 1.2(a)), and then the weakest connections are dropped by setting the lowest-value weights to zero (Figure 1.2(b)). Pruning away these connections is often effective because neurons essentially compute a weighted sum of their input connections, and connections with very low weight values contribute very little to the neuron's overall output. In this example, approximately half the weights in the network have been removed and the DRAM storage needed for the weights is accordingly halved. When the network is now used to classify an input, the bandwidth required to load the weights from memory, and the associated power cost, are also cut in half.

## Power During Training

Training consumes substantially more energy than inference. A single training iteration using stochastic gradient descent on a single sample (e.g., an image) takes approximately three times the number of weight accesses and operations compared to inference on the same sample, as each weight must be retrieved once during the forward pass, retrieved again during the backward pass, and updated to add the newly computed gradient. In addition, training typically takes many thousands of iterations on batches of tens to hundreds of samples each. For example, training the WRN-28-10 image classification network processing 60 images per second would consume a minimum of 15W of power including activations[3] — far beyond the cooling capacity of the iPhone design, which is 5W [33].

[3]WRN-28-10 has 12 million activations, which cost 15.6mJ to access, and must be accessed twice during training. Weights must be accessed 3 times, and computational requirements are roughly 3 times higher. $((47mj + 27mj) * 3 + 15.6mj * 2) * 60s^{-1} = 15.2W$

The proportion of memory bandwidth taken up by weights depends on the network being trained, the training library used, and the underlying hardware. For example, in a multi-layer perceptron (MLP), weights are not reused within a single image, and will take up a larger proportion of the bandwidth; on the other hand, in a convolutional network, each weight can be reused multiple times, so, underlying hardware permitting, less bandwidth is needed.

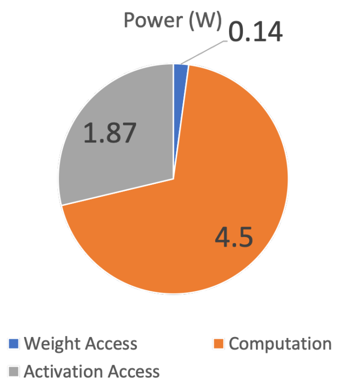
Figure 1.1 shows the power breakdown for two hypothetical libraries training WRN-28-10, a convolutional network (i.e., the worst case for weight pruning). Figure 1.1(b) describes an extreme points where weights are never reused per image in a batch, and so must be accessed and updated for each activation map pulled from memory (this is similar to other network architectures such as MLPs). Figure 1.1(a) shows the opposite extremum, where the library and underlying hardware allow perfect reuse, and the absolute minimum number of weight accesses occur in each mini-batch[4].

In the worst case — training a convolutional network, with perfect weight reuse, perfect hardware, and optimal scheduling — activations can take up to 13× more power than weight accesses[5]. However, activations can be readily compressed: solutions such as gradient check-pointing [7] can compress activations by as much as 7× by trading off fewer memory accesses for extra computation. In addition, activations dominate only when many training examples are grouped together in large batches for parallelism, such as in high-end devices like GPUs or TPUs; in smaller devices with less compute resources, training with single images is common and effective [47], and weights again dominate as in Figure 1.1(b). For WRN, it is possible to have more weights than activations if the input images are small, in this case 32x32x3.

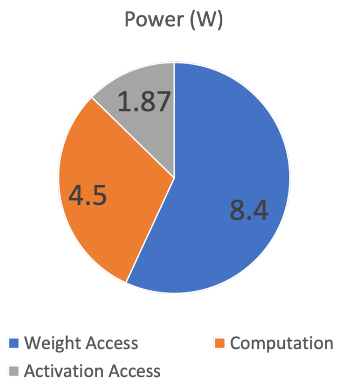
On-device training therefore requires dramatic improvements in the number of weights stored and accessed during the training process. Unfortunately, as we discuss below, existing pruning techniques cannot be applied to reduce training-time weight counts.

[4]This was calculated using the same equation as the initial example in this section, footnote 3, modified so weights are only accessed once per mini-batch of 60 images, with one mini-batch being processed per second: $(27mj * 3 + 15.6mj * 2) * 60s^{-1} + 47mJ * 1s^{-1} = 6.8W$

[5]$1.87/.14 = 13.36$

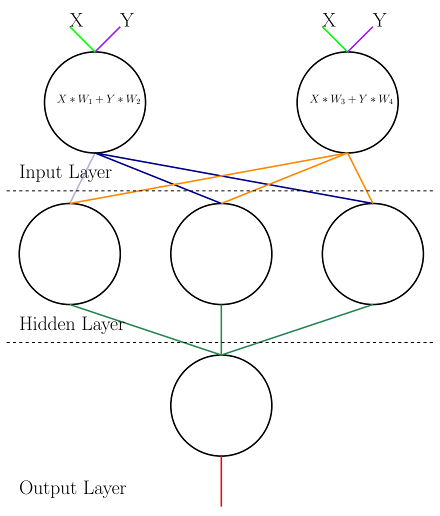


**(a)** The full weight reuse library.

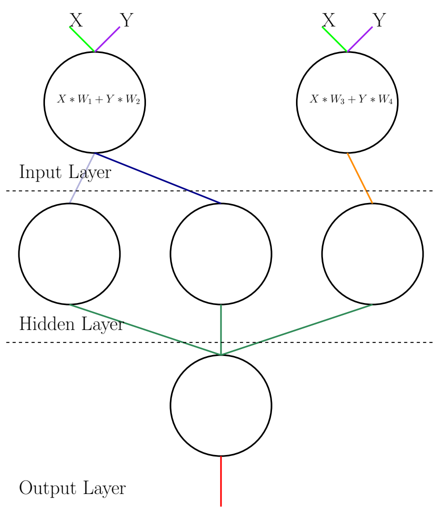


**(b)** The no-weight reuse library.

**Figure 1.1:** Hypothetical power usages while training WRN-28-10 with a batch size of 60, with iterations taking one second (60 images per second).



**(a)** The unpruned MLP.



**(b)** The pruned MLP.

**Figure 1.2:** A Basic three layer multi-layer perceptron (MLP), unpruned in (a) and pruned in (b).

**Limitations of Existing Pruning Techniques**

Existing pruning techniques [e.g., 38] require that the entire network with all weights be fully trained before the pruning step can begin. Typically, the lowest-valued weights are then removed from the network (i.e., set to zero), which often results in an accuracy drop; to counteract this, the pruned network is then re-trained to recover some accuracy [21].

Because existing pruning approaches start with a fully-trained unpruned network, they cannot be used directly during training. It is possible, however, to imagine a scheme where this weight-magnitude-based pruning stage is applied during every iteration, and the full set of weights is never stored. We examine this possibility in Section 3.1, where we show that magnitude-based pruning fails to attain acceptable accuracy in nearly all cases. This is because in the beginning stage of training all weights are initialized using random values [37], and a weight with low initial value can increase dramatically before the training process converges.

Consequently, reducing the weight storage needs and the number of weights accessed during training — and thereby reducing the energy usage and power requirements — calls for a novel approach to weight storage during training. We briefly outline our approach below.

## 1.2 DropBack: Pruning While Training

In this thesis, we develop DropBack, a novel pruning algorithm that (a) can train deep neural networks without accuracy loss while storing up to 4.5× fewer weights during the training process, and (b) produces a pruned network with weight reduction comparable to state-of-the-art post-training pruning techniques on modern networks.

DropBack is based on three key insights: (1) that weights that have accumulated the most gradient updates over time account for most of the learning; (2) that accumulated gradients are predicated on how the initialization values are chosen for the remaining weights; (3) that a pseudo-random number generator can recompute initialization values as required and do not need to be stored. Based on these observations, DropBack stores only critical weights during training and recomputes the initial value for the rest of the weights, called untracked weights. Intuitively, recomputing the non-critical weights allows the training algorithm to leverage the scaffolding created by their initial values to improve the learning process for the critical weights. The DropBack algorithm is described as Algorithm 2 in Chapter 3.

The weight reduction substantially reduces both memory footprint and memory bandwidth requirements during training. For example, DropBack can reduce the active weight count in the convolutional neural network WRN over 360×, matching the baseline performance of VGG-S, reducing the required memory bandwidth from 8GB/s to 22MB/s when labeling images at 60 images per second. The network weights now take 0.4MB to store, comfortably fitting in the on-chip SRAM memory of a recent iPhone [33], where accessing 64 bits of on-chip data costs only 5pJ on a 45nm process (instead of 2.6nJ off-chip) [25]. Interestingly, the DropBack algorithm preferentially removes entire $3 \times 3$ convolutional filters (up to 99%), even though the algorithm itself has no concept of a convolutional filter.

DropBack differs substantially from prior pruning techniques, which either (a) train an unconstrained network, prune the network, then retrain the network, or (b) add an additional term to the loss function of the network to encourage sparsity that can be used for post-training pruning. DropBack instead prunes the network from the very first iteration, does not require separate pruning and retraining phases, and does not require additional modifications to the loss function of the network.

DropBack outperforms best-in-class pruning methods on network architectures that are already dense and have been found particularly challenging to reduce [41–43]. On Densenet, DropBack achieves 5.86% validation error with 4.5× weight reduction, and on WRN-28-10 DropBack achieves 3.85%–4.20% error with 4.5×–7.3× weight reduction. For both Densenet and WRN-28-10 these weight reduction ratios are state of the art compared to post-training pruning techniques. The memory used during training and inference for weight storage is limited, and no extra steps are required to reduce the network after training.

With the addition of an initial weight decay parameter the untracked weights can be reduced to zero, providing *both* memory and computational sparsity. On Densenet with 500K tracked weights in Section 4.4.2, DropBack with decay can achieve 77.77% sparsity — i.e., 77.77% of the weights are zero and can be removed entirely — without any accuracy loss. On WRN-28-10, DropBack with decay can achieve 86% sparsity (500K tracked weights) without decrease in accuracy, and 99% sparsity with only a 1% increase in error (100K tracked weights). The weight decay variant of DropBack is described as Algorithm 3 in Section 3.3.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 contains background information on neural networks and related work, including other methods of reducing the memory footprint and energy consumption of neural networks. Chapter 3 presents the intuition behind DropBack and develops the DropBack algorithm in three phases. Chapter 4 reports and discusses the accuracy and weight reduction ratio of the final two DropBack variants, and presents relevant tradeoffs. Finally, Chapter 5 discusses potential future research directions and concludes this work.

# Chapter 2

# Background

Understanding DropBack requires a basic knowledge of neural networks, how they are trained, and how they can be compressed using existing techniques. This chapter lays out the two types of neural network used in this thesis — multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs). It then discusses how the computational and memory requirements of these networks differ during training and inference, and describes the different compression methods currently in use.

## 2.1 Neural Networks

Neural networks are constructed of many interconnected neurons, each of which performs a simple mathematical function. In the two variants we consider here, information flows across the network in a feed-forward fashion, with the outputs of neurons in earlier "layers" becoming inputs to neurons in later layers. Neurons organized in regular structures and trained with large datasets have achieved state-of-the-art performance in problems such as classification, regression, and speech understanding. By convention, networks are called "deep" if they contain more than several layers.

Before discussing how to reduce the computational and memory impacts of training, we first give a basic outline of their operation. Section 2.1.1 describes the multi-layer perceptron, one of the simplest forms of neural networks. Section 2.1.2 describes how neural networks are used once trained, and section 2.1.3 gives a basic overview of how neural networks are trained.

### 2.1.1 The Structure of MLPs and CNNs

The simplest neural network consists of one neuron, and is called a *perceptron*; such a network is shown in Figure 2.1. The perceptron takes two inputs, $x$ and $y$, and outputs a single resulting value, $f(x, y)$. Internally, the perceptron stores constant two weights, $w_1$ and $w_2$, one for each input. The mathematical operation performed by the perceptron is the dot product, so we have $f(x, y) = w_1 x + w_2 y$. Typically the output of a perceptron is then passed through an *activation function* — originally the Heaviside step function [54], and more recently the sigmoid [37] and rectified linear unit (ReLU) [1] functions. ReLU is typically used in vision-related applications [1], and merely cuts off all negative values at 0:

$$f(x) = \max(0, x) \tag{2.1}$$

Perceptrons can be combined into multiple layers to create a multi-layer perceptron (MLP). An example of a simple three-layer MLP is shown earlier in Figure 1.2(a). The input (topmost) layer takes two inputs, $x$ and $y$, and feeds each to two different neurons (i.e., perceptrons). Each neuron has its own weights for both $x$ and $y$, labeled $w_1$, $w_2$, $w_3$, and $w_4$. Each neuron has a single output, $o_1$ and $o_2$ respectively, which is the dot product of their respective weights followed by an activation function. In this example, the activation function is ReLU (Equation 2.1). The outputs of the input layer are the inputs to the hidden layer of three neurons, with each neuron in the hidden layer receiving $o_1$ and $o_2$ as inputs. The hidden layer passes outputs to the final (output) layer, a single neuron.

Compared to a single perceptron, MLPs are capable of solving non-linearly separable problems, and, indeed, any computable function [12]. However, because every neuron has a separate weight for every feature in the input, they quickly grow too large to train even with a modest number of input dimensions. For example, in the commonly used ILSVR2012 dataset [55], images are typically scaled to 256x256x3, for a total input size of 196,608 pixels; given an MLP with 1,000 neurons in the input layer, just that layer would use approximately 800MB of weights. In order to scale neural networks to problems with many inputs, such as image recognition, alternative structures are therefore used.

The most common alternative to basic MLPs is a convolutional neural network (CNN). CNNs are built out of convolutional layers, pooling layers, and typically one or two “dense layers” — another name for a layer of an MLP — as the final layers. The convolutional layers take advantage of the observation that vision tasks — such as detecting edges — are the same regardless of where in the image they occur, i.e., are invariant under translation. The convolutional layers are therefore built from convolutional “filters” which, when convolved with a set of neighboring pixels, produce one output value. The operation of the filter is shown in Figure 2.2. A filter will only consider pixels in its receptive window — e.g., the filter in Figure 2.2 has a 3x3 receptive window — and pixels outside this window never contribute to the current output value being computed. To produce an entire output layer, the convolutional filter is moved across the input layer (e.g., left to right in Figure 2.2). Typically, each convolutional layer consists of many filters, the outputs of which are stacked to produce many output feature maps as inputs to the next layer. Pooling combines the outputs from a few neurons, usually using the max function to effectively downsample the previous layer. As with the “dense” MLP layers, convolutional and pooling layers are stacked together to form a network. Figure 4.1 shows a fairly small CNN which we use as one of the evaluation models.

### 2.1.2 Inference

For the inference task, such as classifying an input image as a cat vs. a dog, the neural network is run forward: inputs are provided to the first layer, which in turn computes the inputs to the second layer, and so on. The outputs of the final layer form the output of the entire network; for example, they might represent confidence that the image belongs to a certain class (e.g., cat or dog).

During a forward pass, the weights of the network are only read and never updated, and storage needed for intermediate results is limited to a single layer.[1] The amount of intermediate result memory required to perform inference can therefore be minimal, depending on the exact structure of the network and how the computations are scheduled. On the other hand, all of the weights for the entire network must be

[1]Or a small, constant number of layers in more advanced networks like residual networks [22]

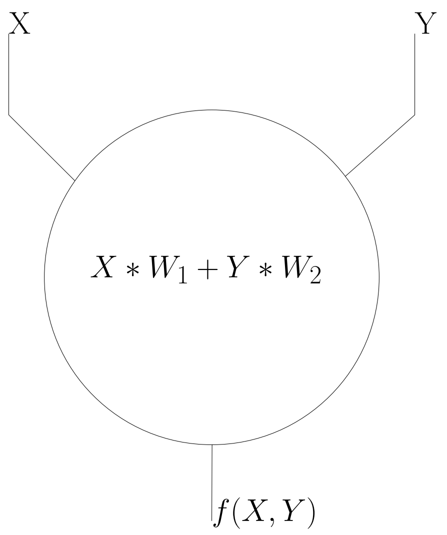


**Figure 2.1:** A basic two input single output perceptron.

read during each inference pass.

The amount of computation and memory required for inference tasks with a given network can often be further reduced by modifying the network through pruning, which removes some weights from the network, or quantization, which reduces the number of bits needed to represent each weight. Pruning is described in Section 2.2.1 and quantization in Section 2.2.2.

### 2.1.3 Training

To train a neural network, the weights of each neuron are first initialized to a small random value, usually drawn from a scaled normal distribution [40]. The aim of training is to change these weights to give the correct output for each example in the training set, which contains multiple inputs, each paired with its respective "ground truth" label. If the training set is drawn from the same distribution as the examples the network is expected to classify, and care is taken to prevent the model from overtraining, a trained network can be deployed and perform with accuracy close to its performance on the training set.

Weights are typically trained using an optimization algorithm called stochastic gradient descent (SGD). First, after one or more inputs (a "batch") are classified

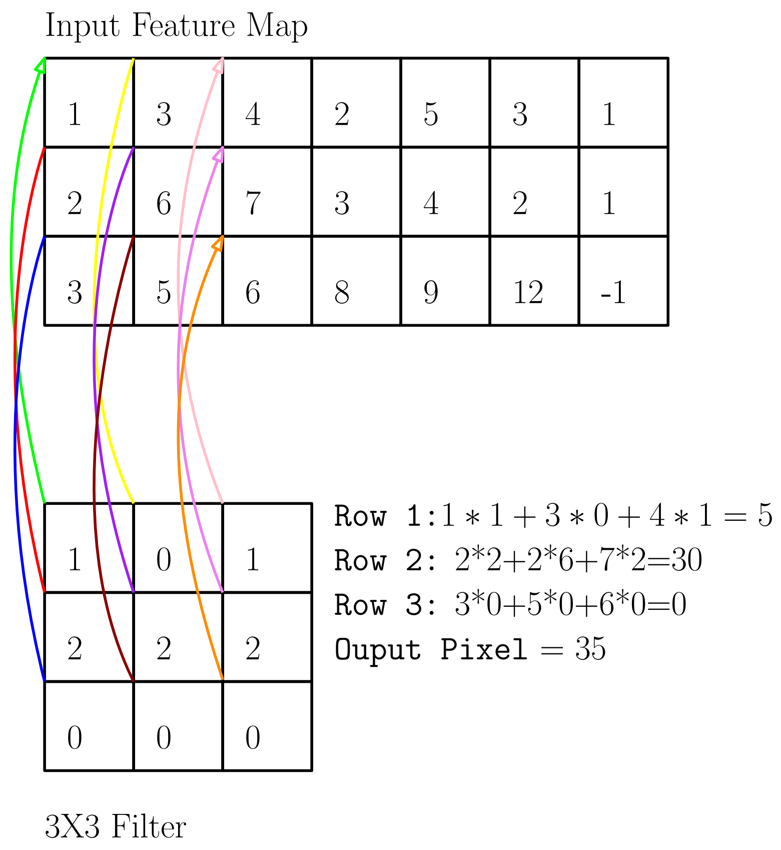


**Figure 2.2:** A 3x3 convolutional filter calculating the value of a single output. To compute the entire output feature map the filter would be strided along the input feature map, to produce five total outputs this example.

using the forward pass, the loss (prediction error) on these inputs is computed; the mean squared error, shown in Equation 2.2, is commonly used as the loss function. Next, the algorithm calculates the gradient of each weight with respect to the loss. Finally, each weight's gradient is scaled by a factor called the learning rate and subtracted from the weight as shown in Equation 2.3. In effect, each weight is updated based on how much it contributed to the error of the current output compared to the ground truth.

Applying the chain rule to the problem of computing any gradient in the network yields an algorithm called back-propagation, or backprop [37]. The key observation is shown in Equation 2.4: the partial derivative of loss with respect to the weight can be chained through the unactivated output of the neuron itself and through the activation function by multiplying the relevant partial derivatives. Because this can be done in a layer-by-layer fashion starting from the last layer and ending on the first, the weight-update part of the backprop algorithm is also known as the backward pass.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \tag{2.2}$$

$$\Delta w_i = -\alpha \frac{\partial E_{MSE}}{\partial w_i} \tag{2.3}$$

$$\frac{\partial E_{MSE}}{\partial w_i} = \frac{\partial E_{MSE}}{\partial a_i}\frac{\partial a_i}{\partial net_i}\frac{\partial net_i}{\partial w_i} \tag{2.4}$$

The backward pass is far more memory intensive than the forward pass; the exact factor depends on the implementation details of the deep learning library used and the network being trained, but can range from 2× to 100×. A significant component of this cost comes from the need to store the activations of each layer, needed to compute how the loss changes with respect to each weight. For an MLP, this activation storage cost is equivalent to the number of weights in the next layer; for a CNN, on the other hand, the number of activations for each weight is much higher since the same small set of weights (i.e., filter) produces an entire output map.

Compressing or reducing the number of activations in CNNs and neural networks is therefore an active area of research. Some research has focused on storing activations in larger, slower memories farther away from the computation such as [52], while another work focused on recomputing the activations as the backward pass progresses [7]. Activations have also been compressed successfully through omitting zero values [46, 53, 74]. While activation storage is expensive, activations are only ever accessed twice — once to write the activations from the forward pass, and once to read them for the current neuron in a backward pass. In contrast, weights are repeatedly accessed— especially in CNNs — during the forward pass, and during the update from the backward pass.

DropBack complements the work on reducing activation storage at training time by also reducing storage requirements for weights during training. Chapter 3 describes how DropBack modifies the training process in a way that can reduce the number of weights that need to be stored by an order of magnitude or more.

## 2.2 Compression Techniques

Research to date has examined two main methods of reducing the memory and computational overhead of deep neural networks: pruning and quantization. Pruning a neural network removes weights, filters, or entire sections of the network deemed unimportant to producing network's final output. Weights to be removed are selected via a heuristic, e.g., removing the smallest-magnitude weights. Quantization, on the other hand, does not modify the *structure* of the network itself but instead changes how its weights and activations are *represented*. Each value is stored using fewer bits than normal (which means that the stored value is approximate), reducing both the memory needed to store the weights and/or activations as well as the computational cost. Other techniques have been proposed such as HashedNets [8] and Huffman coding [18], but are less commonly used than pruning or quantization as they require significant restructuring of neural network libraries.

### 2.2.1 Pruning

Pruning, or the removal of specific parameters in a neural network, was introduced in 1990 as a technique for improving generalization and reducing the number of examples needed to train a network [38]. Magnitude-based pruning is the most basic approach: it simple removes the lowest-magnitude weights up to a user-specified percentage (e.g., 95% of the weights). Like most pruning methods, magnitude-based pruning requires that the network be trained first, then pruned, and only then retrained to recover accuracy. This method was effective on the smaller datasets used by LeCun et al. [38], but loses significant accuracy compared to more modern methods on larger networks and datasets. While this technique analytically predicts the effect of pruning and prunes weights with the smallest predicted perturbation [38], the retraining step is still required.

Later work considered using the second derivatives to select parameters to prune [21], or training the network itself to learn which connections are important through either $\ell^1$ or $\ell^2$ regularization [18]. Computing the second derivatives of all the parameters is, however, computationally and memory intensive, and training with $\ell^1$ and $\ell^2$ regularization results in slower training. Alvarez and Salzmann [3] focused on reducing the rank of the parameter matrices during training for later compression, again at the cost of increased time required for training, and is not able to reduce training time memory usage compared to standard training.

Overall, pruning networks to enforce sparsity levels of 90%–99% after training has been effective for many networks [14, 18, 38, 44, 46, 60, 64, 65, 74]. In contrast to DropBack, all of these post-training pruning techniques require a retraining step to regain lost accuracy, and both pruning and low-rank constraints still require un-pruned backpropagation during the training phase. In fact, the memory and energy cost of the extra backpropagation step in standard pruning methods *increases* the barrier to training in low-power embedded systems.

Other work has attempted to prune the network *during* training, either to improve accuracy or to increase sparsity. Zhu and Gupta [75] gradually increase the number of weights masked from contributing to the network, while Molchanov et al. [49] extend variational dropout [32] with per-parameter dropout rates to increase sparsity. Babaeizadeh et al. [4] inject random noise into a network to find and merge the most correlated neurons. Finally, Langford et al. [35] decay weights every $k$ steps (for a somewhat large value of $k$), inducing sparsity gradually. However, unlike DropBack, all of these techniques store the entire unpruned network (plus potentially a $2\times$ overhead for extra state per weight in the network) and therefore require at least as much memory to train as the unpruned network.

### 2.2.2 Quantization

Quantization can be used to reduce the inference-time and training-time memory and compute requirements by representing the network weights and activations with lower precision than normal. To reduce storage costs, numbers can be quantized after training [9, 14, 17, 18, 64, 68, 72], or during an iterative retraining process [72]. All of these techniques quantize floating-point number representations to fixed-point representations; this can reduce the energy required for *computation* as floating-point operations are more energetically expensive than fixed-point operations. Both Zhou et al. [72] and Gysel [17] go further and either encourage or mandate weight values to be powers of two; this allows for multiplications to be replaced by single shift operations, saving computation energy. Importantly, while the final product of these methods is quantized, they all rely on the full floating-point weights being available during training.

Quantization can also performed during training instead of as a post-processing step as shown by Cai et al. [6], Courbariaux et al. [10, 11], Gupta et al. [16], Holt and Baker [24], Hubara et al. [28], Mishra et al. [48], Rastegari et al. [51], Simard and Graf [57], Zhou et al. [73]. Out of all of these methods, only Courbariaux et al. [10], Gupta et al. [16], Hubara et al. [28], Mishra et al. [48] use reduced precision while training to lower the training-time storage costs; other methods store the full non-quantized weights during backpropagation just like the post-training methods. Quantization is orthogonal to pruning (and, in particular, to DropBack), and the two techniques can potentially be combined.

Quantization has also been used to reduce the overhead of gradient broadcasts in distributed training environments. Seide et al. [56] reduce gradients to a single bit upon broadcast, saving 32× the bandwidth compared to standard 32-bit floating-point (FP32) gradient broadcasts. Similar approaches with less quantization than that achieved by Seide et al. [56] have been studied by Alistarh et al. [2], Wangni et al. [66], Wu et al. [67]. All of these works have a shared limitation. Each local device maintains the full FP32 parameters as well as an additional error term for each gradient to correct the quantization upon each broadcast, so such approaches reduce inter-node communication costs at the cost of local storage. Because DropBack only ever tracks a small subset of gradients, a distributed implementation could also reduce broadcast costs (by the same amount as the weight compression factor) by only syncing the tracked weight sets, and doing so *without* maintaining a full set of the parameters on each local training node.

### 2.2.3 Other Compression Methods

Very few techniques have attempted to reduce the inference-time and training-time memory and compute costs. HashedNets [8] use a hash function to group neuron connections into buckets; within each bucket, all connections use the same weight value. In effect, HashedNets quantize the weights, but use a lookup table to permit a full 32-bit floating-point weights to be used during computation. However, the technique was never tested beyond MNIST, a small and extremely easy dataset, where it achieved decent but not state-of-the-art compression.

Huffman coding [29] has also been used to compress neural networks [18]. As a lossless compression technique, the coding was applied *after* both pruning and quantization had been performed, and after the network had been trained to completion. This allowed the network to be compressed approximately 2× more than pruning and quantization alone, but required additional computation to decompress the weights on every access, and complicated the memory access pattern. Huffman decoding is far more expensive than the weight regeneration mechanism used to represent non-tracked gradients in DropBack (see Figure 3.2), but is orthogonal to our technique and can be used to compress the tracked weights if the energy cost is warranted.

## 2.3 xorshift

The xorshift algorithm [45] is a very lightweight method of generating high quality pseudo-random numbers. Shown in Algorithm 0, the state $x$ is initialized to any real integer, $\mathbb{Z}$\$, and then shifted and xor-ed three times. Later, in Section 3.2, this xorshift algorithm will be modified to regenerate initial values.

**Algorithm 0:** The 32-bit XORSHIFT algorithm. The parameter $x$ is a single 32-bit non zero number; for deterministically regenerating weights, $x$ is the flattened index of a weight plus a constant factor that is different for every training run.

```
Initialization: $x \sim \mathbb{Z}$
Output: $x$
$x = x \oplus (x << 13)$
$x = x \oplus (x >> 17)$
$x = x \oplus (x << 5)$
```

## 2.4 Summary

DropBack differs from all of these techniques by specifically targeting memory constraints during training, and, unlike all of them, prunes the network from the very first iteration of training. DropBack is described next in Chapter 3.

# Chapter 3

# The DropBack Algorithm

Uniquely among pruning techniques, DropBack prunes weights from the very first iteration of training; the full set of weights is *never* stored in or retrieved from memory. To provide this reduction in memory usage, DropBack continuously omits weights that it determines to be least important to the final output of the network; these are called untracked weights. The final version, Algorithm 3, produces a sparse network where all the untracked weights are zero during inference.

This chapter develops DropBack in three steps, each building on insights from the previous version:

- Algorithm 1, where untracked weights are set to zero.
- Algorithm 2, where untracked weights maintain their initial value which can be regenerated on the fly without storage.
- Algorithm 3, where untracked weights are decayed from their initial value to zero over time during training.

The rest of this chapter describes each of these in detail.

## 3.1 Algorithm 1

We first investigated an adaptation of the magnitude-based pruning approach [38, etc.] commonly used for post-training pruning. In this algorithm, each iteration of training tracks only the highest-magnitude weights, while all other weights are set to zero. The details of the computation are shown in Algorithm 1. In the algorithm,

$W_{trk}$ and $W_{utrk}$ represent tracked and untracked weights, $T$ and $U$ are tracked and untracked accumulated gradients, $S$ is the set of sorted accumulated gradients, $k$ is the number of gradients to track, $\lambda$ is the lowest tracked cumulative gradient, and $\alpha$ is the learning rate. The *mask* indicates a boolean matrix with the same shape as the weights, and $\overline{mask}$ indicates its logical inverse. The $\mathbb{1}$ operator indicates that the mask is boolean.

Although the listing in Algorithm 1 sorts all the weights before pruning for exposition clarity, it is not necessary to physically store and sort the full weight set. In a practical implementation, the tracked weight set would be stored using a priority queue of size $n$, in which an incoming gradient higher than the stored minimum evicts the smallest element.

In contrast with the common post-training magnitude based pruning techniques, Algorithm 1 encapsulates the training-pruning-retraining phases in a single training step, which reduces the number of epochs required to converge. In essence, each step in this algorithm is equivalent to a post-pruning retraining step. Ideally, this would move the network towards an optimum while reducing the number of parameters from the very first training step.

We tested Algorithm 1 using both LeNet-300-100 [37] and a smaller multi-layer perceptron with only 100 hidden neurons, which we refer to as MLP-100. The achieved weight reduction ratio was slightly less than 2×, an inferior result compared with other existing pruning techniques (detailed results are discussed in Section 4.2). Below, we investigate why Algorithm 1 performs poorly, and draw insights for creating a better version.

### Initial Weights are a Poor Metric for Selection

Proper weight initialization is essential in order to train deep neural networks quickly, and even to learn at all [40]. These initial weights are typically drawn at random from a scaled Gaussian distribution, and updated by the stochastic gradient descent optimization algorithm. During the first iteration of training, weight updates are scaled by a typical learning rate of 0.001 to 0.1 when using stochastic gradient descent with momentum, or higher learning rates of up to 0.5 without momentum, so they move little compared to their initial magnitude. As a result, pruning away the lowest-magnitude weights after the very first training step leads to a nearly random selection of weights to be dropped. In effect, the only weights that are tracked are those with the highest initial value, and no other weights have the opportunity to learn.

#### Vanishing Gradients Result in Poor or No Learning

Algorithm 1 suffers from another problem that stems from immediately setting untracked weights to zero. When most of the weights are zero, the activations in much of the network also become zero during the forward pass. In turn, this causes the gradients of the loss with respect to those weights to also become zero during the backward pass, which inhibits learning. In our experiments, for example, setting the 90% of the MLP-100 network to zero (10$\times$ weight reduction) caused nearly 99% of the gradients to become zero as well, preventing optimization.

This effect is well-known in the literature as the vanishing gradient problem [15]. To counteract vanishing gradients, the post-training pruning schemes such as those of Zhu and Gupta [75] and Han et al. [18] increase the sparsity of the network very gradually so that relatively few additional weights are zeroed in every training iteration; although the final sparsity achieved with those methods can be as high as 90%, that level of sparsity is only achieved at the very end of training. Because we aim to reduce the memory footprint from the very start of the training process, however, we cannot employ this gradual sparsity technique in DropBack.

## 3.2 Algorithm 2

The second step in developing DropBack is based on the two key observations from evaluating Algorithm 1: (a) that dropping the lowest-magnitude weights early on inhibits learning, and (b) that resetting weights to zero results in the vanishing gradient problem. We take advantage of these observations in the three insights discussed below.

#### Insight 1: Track the Highest Accumulated Gradients

Section 3.1 shows that the naive approach of tracking only the highest-magnitude weights is not effective during the first few training iterations, and so cannot be used

**Algorithm 1:** Pruning the Lowest-Magnitude Weights

```
**Initialization:** $W^{(0)}$ with $W^{(0)} \sim N(0, \sigma)$
**Output:** $W^{(t)}$
**while** *not converged* **do**
    $T = \left\{ \left| W^{(0)} + \sum_{i=0}^{t-1} \frac{\alpha \partial f(W^{(i-1)}; x^{(i-1)})}{\partial w} \right| \text{ s.t. } w \in W_{trk} \right\}$
    $U = \left\{ \left| W^{(0)} + \frac{\alpha \partial f(W^{(i-1)}; x^{(i-1)})}{\partial w} \right| \text{ s.t. } w \in W_{utrk} \right\}$
    $S = \text{sort}(T \cup U)$
    $\lambda = S_k$
    $mask = \mathbb{1}(S > \lambda)$
    $W^{(t)} = mask \cdot \left(W^{(t-1)} - \alpha \nabla f\left(W^{(t-1)}; x^{(t-1)}\right)\right) + \overline{mask} \cdot 0$
    $t = t + 1$
**end**
```

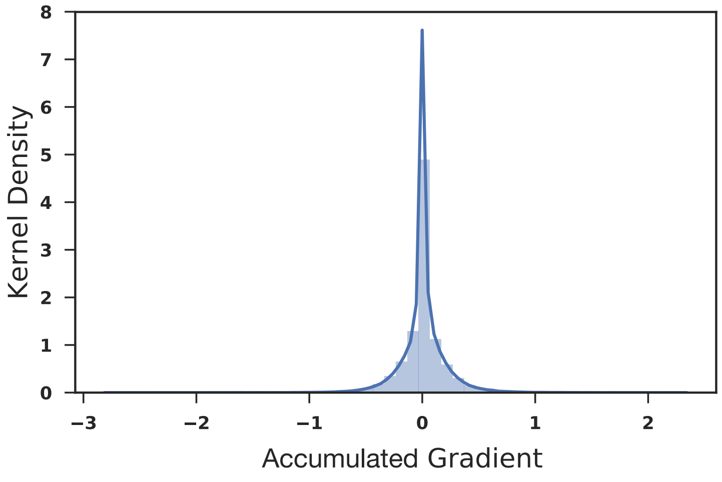


**Figure 3.1:** Distribution of accumulated gradients over 100 epochs of standard SGD training on MNIST using a 90,000-weight MLP.

to reduce the memory footprint throughout the entire training process. Informally, this is because the initial value of each weight, typically drawn from a scaled normal distribution [40], serves as scaffolding which gradient descent can amplify to train the network.

Instead, our approach in DropBack is to first observe that each weight is a sum of its initial value and the gradient accumulated during training, and logically separate the two components. (At first sight, this may appear counterproductive because now the initial value may need to be stored for each weight; we resolve this problem using Insight 2 below.) Then, to reduce the number of weights tracked, we keep only a fixed number of weights which have *learned the most overall* — that is, the weights with the highest accumulated gradients.

To validate this hypothesis, we examined the distribution of accumulated gradients during the first 100 training iterations of a 90,000-weight MLP on the MNIST dataset (see Section 4.1 for experimental method details). In the histogram, shown in Figure 3.1, most gradients are very close to zero. This shows that weights move very little from their initial values, and suggests that only a small fraction of gradients needs to be tracked *provided* the remaining weights are kept at their initial values.

#### Insight 2: Recompute Initialization-time Values for All Untracked Weights

As discussed in Section 3.1, simply setting untracked weights to zero impairs training; this is because the scaffolding provided by the initialization values is critical to the accuracy of the trained network in order to prevent vanishing gradients. If the initialization values can be preserved, we reasoned, we should be able to achieve higher accuracy and better pruning.

To validate this hypothesis, we trained the 90,000-weight MLP on MNIST as in Section 3.1, but this time allowed the untracked weights to retain their initialization-time values. We observed that the tracked weights could be reduced up to 60× if initialization values were preserved, but only 1.8× if untracked weights were zeroed (see Section 4.2.1 for details). This is in line with the observation that zeroing weights causes gradients to vanish (see Section 3.1), and suggests that initialization-time values for all weights should somehow be preserved.

However, storing the initialization-time values would require accessing memory to retrieve them during both the forward and backward passes of training — a costly proposition for large networks where an off-chip memory access consumes upwards of 2900× more energy than a 32-bit floating-point add.

To avoid storing these initial weights, we observe that in practice the values are initialized using a pseudo-random number source that is initialized using a single seed value and post-processed to fit a scaled normal distribution [40]. Because each value only depends on the seed value and its index, it can be deterministically regenerated precisely when it is needed for computation, without ever being stored in memory.

While the value might still be very briefly stored in the on-chip register file (in a CPU or GPU), register files are small SRAMs located very close to the processor's functional units, and accessing them costs a fraction of the energy needed to access off-chip DRAM [25], for example. In addition, as the register file only serves to communicate values between individual CPU or GPU instructions, the lifetime of this storage is limited to the few instructions that it takes to generate the value and use it to multiply with a single activation value, so only a very small amount of storage is needed.

To recompute a normally distributed pseudo-random initialization value, we employ the XORSHIFT algorithm described earlier in Section 3.2, modified to output a floating-point number between $-1$ and 1. This computation, shown in Algorithm 4, requires eight 32-bit integer operations and one 32-bit floating-point operation. Recomputing the weight, therefore, consumes about 1.5pJ of energy in a 45nm process, which is still 1700× less energy than a single off-chip memory access.

Regenerating untracked parameters to their initial values also works out-of-the-box for layers like Batch Normalization or Parametric ReLU, where the initialization strategy is typically a constant value that is the same for the entire network. In this case, DropBack merely regenerates the constant value rather than using Algorithm 4. As a result, these layers are also pruned by DropBack, a unique feature not found in other pruning methods.

**Algorithm 4:** The 32-bit XORSHIFT algorithm modified for generating random floating-point numbers between $-1$ and 1. The parameter $x$ is a single 32-bit non zero number; for deterministically regenerating weights, $x$ is the flattened index of a weight plus a constant factor that is different for every training run. The output of this algorithm can then be scaled by the number inputs to a layer as suggested by LeCun et al. [40].

```
**Initialization:** $x \sim \mathbb{Z}$
**Output:** $x$
$x = x \oplus (x << 13)$
$x = x \oplus (x >> 17)$
$x = x \oplus (x << 5)$
$x = (x \& \texttt{0x007fffff}) | \texttt{0x40000000}$
$x = x - 3.0$
```

## Insight 3: After a Few Epochs, Freeze Which Weights Are Tracked

During training, it may happen that a "new" gradient for an untracked weight exceeds one of the accumulated gradients tracked by DropBack. This is very common during the initial phase of training, and is an artifact of the optimization algorithm's efforts to select the most productive direction. However, this also means that gradients for all of the untracked weights are still computed in every iteration. Selecting the tracked set costs additonal energy as well, as either software or hardware sorting is required. While the energy required for gradient computation and comparison is small compared to retrieving all weights from off-chip DRAM, it can noticeably contribute to the total energy expenditure once only a small fraction of weights are tracked.

In order to sidestep most of these accesses, we reasoned that the weight selection should stabilize after a few epochs. By then, most of the tracked gradients should have accumulated sufficiently large magnitudes that it is unlikely that they would be exceeded by a "new" gradient for an untracked weight; indeed, as most accumulated gradients have small magnitudes (see Figure 3.1), we expected most weights to move little.

To validate this intuition, we trained the 90,000-parameter MLP-100 on MNIST using standard SGD while keeping track of the 2000 weights that had accumulated the highest gradients. Figure 3.2 and Figure 3.3 show that the set of the 2,000 highest-

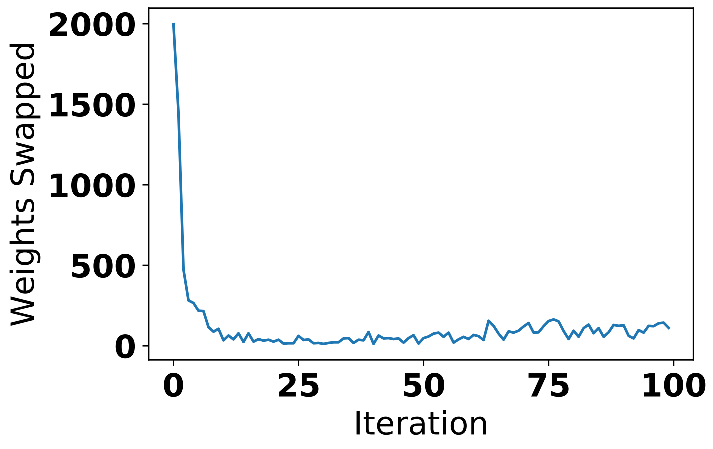


**Figure 3.2:** Number of weights added/removed to the top-2K gradient set in the first 100 iterations on MNIST.

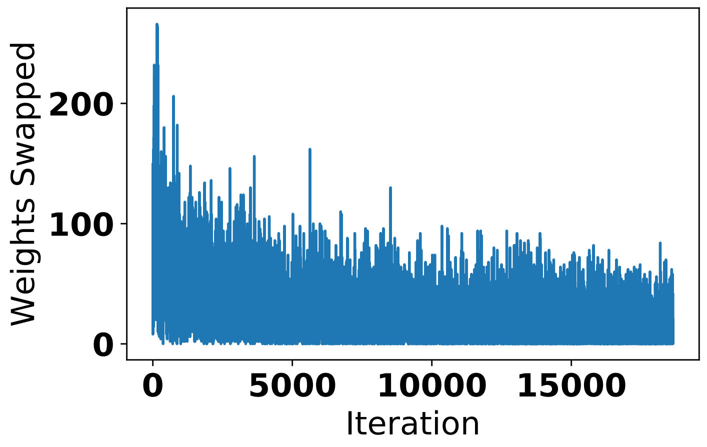


**Figure 3.3:** Number of weights added/removed to the top-2K gradient set after the first 100 iterations on MNIST. Note the y-axis scale is 1/10th the scale of Figure 3.2.

gradient weights stabilizes in relatively few iterations. Before the first iterations, many weights are added to and removed from the set of 2,000 highest accumulated gradients (Figure 3.2). After the first ten iterations, however, the top-2,000 gradient set settles down with $< 0.04\%$ of all weights being added to or removed from the top-gradient set in any iteration (Figure 3.3). This "noise" remains throughout the training process.

This observation allows us to "freeze" which gradients are tracked after a small number of epochs (e.g., 10–20), where an epoch is a full pass through the training dataset (approximately 500 iterations per epoch). After freezing, no new gradients may replace those in the currently tracked set, so gradients need to be computed only for the weights in the tracked set instead of for all weights.

We investigated several methods of selecting the freeze epoch, including via a hyper-parameter, freezing after some validation accuracy has been reached, and freezing once the number of additions/removals became lower than a small percentage of the tracked weight count (see Section 4.3.1). Because freezing at a constant epoch worked better, we chose to freeze at the best epoch for each test in Chapter 4, determined by sweeping the hyper-parameter space for the freeze epoch.

**The Complete Algorithm 2**

Algorithm 2 shows the resulting DropBack training process, which incorporates the three insights above. $N(0, \sigma)$ is generated from the XORSHIFT pseudo-RNG shown in Algorithm 4.

Weights are initialized from a scaled normal distribution [40], generated by the XORSHIFT pseudo-random number generator. After a user defined epoch, the tracked set is "frozen," and gradients are only updated for the weights already tracked; this saves additional computation time and energy. Note that the tracked gradient set $T$ requires no storage: its elements are recomputed when needed from $W^{t-1} - W^{(0)}$. As in standard stochastic gradient descent, training ends once the network is considered to have converged.

**Algorithm 2:** Pruning the Lowest-Magnitude Accumulated Gradients

```
Initialization: $W^{(0)}$ with $W^{(0)} \sim N(0,\sigma)$
Output: $W^{(t)}$
while not converged do
    $T = \left\{ \left| \sum_{i=0}^{t-1} \frac{\alpha \partial f(W^{(i-1)};x^{(i-1)})}{\partial w} \right| \text{ s.t. } w \in W_{trk} \right\}$
    if not frozen:
        $U = \left\{ \left| \frac{\alpha \partial f(W^{(i-1)};x^{(i-1)})}{\partial w} \right| \text{ s.t. } w \in W_{utrk} \right\}$
    else:
        $U = \{\}$
    $S = \text{sort}(T \cup U)$
    $\lambda = S_k$
    $mask = \mathbb{1}(S > \lambda)$
    $W^{(t)} = mask \cdot \left(W^{(t-1)} - \alpha \nabla f\left(W^{(t-1)}; x^{(t-1)}\right)\right) + \overline{mask} \cdot W^{(0)}$
    $t = t + 1$
end
```

## 3.3 Algorithm 3

Although the DropBack Algorithm 2 developed above reduces the storage required for weights during training, using the trained network requires a modified inference algorithm. This is because the trained network relies on the initial values of the non-tracked weights being recomputed as they have been during training, which is not supported out-of-the-box in existing neural network accelerators. Additionally, the regenerated weights still present a computation overhead during inference.

We therefore examined the possibility of slowly decaying these initial weight values to zero during the training process. We reasoned that, once the tracked weights have been trained, the scaffolding provided by the initial weights is no longer necessary, and can be gradually removed. As the initial weight values slowly decay to zero, the tracked weights — the only weights that change during training — should gradually adjust to compensate for the lower values (and, eventually, the absence) of the non-tracked weights.

To test this intuition, we allowed the initial values to decay to zero over time using an exponential decay term controlled by the number of iterations, each iteration scales down the initial values regenerated until they reach 0. Details are shown in Algorithm 3. In all of our experiments, using 32-bit floating-point weights, every initial parameter has decayed to zero by iteration 1,000. Overall, DropBack with the decay modification achieves the best accuracy to weight reduction tradeoff on all of the networks used in this thesis except the MLPs (see Section 4.2.3 for details).

**Algorithm 3:** Decaying the Initial Parameters

```
Initialization: $W^{(0)}$ with $W^{(0)} \sim N(0, \sigma)$
Output: $W^{(t)}$
while not converged do
    $T = \left\{ \left| \sum_{i=0}^{t-1} \frac{\alpha \partial f(W^{(i-1)}; x^{(i-1)})}{\partial w} \right| \text{ s.t. } w \in W_{trk} \right\}$
    if not frozen:
            $U = \left\{ \left| \frac{\alpha \partial f(W^{(i-1)}; x^{(i-1)})}{\partial w} \right| * \delta^{t-1} \text{ s.t. } w \in W_{utrk} \right\}$
    else:
            $U = \{\}$
    $S = \text{sort}(T \cup U)$
    $\lambda = S_k$
    $mask = \mathbb{1}(S > \lambda)$
    $W^{(t)} = mask \cdot \left(W^{(t-1)} - \alpha \nabla f\left(W^{(t-1)}; x^{(t-1)}\right)\right) + \overline{mask} \cdot W^{(0)} * \delta^{t-1}$
    $t = t + 1$
end
```

## 3.4 Differences with Existing Approaches

Most existing pruning techniques (discussed in Section 2.2.1) either require the network to be retrained after pruning is performed, or need additional terms to be applied to the loss function of the network to encourage increased sparsity. Pruning techniques that rely on retraining are well-studied [14, 18, 44, 46, 64, 74, etc.], and can achieve close to the original accuracy of the unpruned network. Rather than decrease the memory footprint during training, these algorithms actually *increase* it because they typically track the full network plus additional parameters. In addition, they typically increase training time, as an additional retraining or fine-tuning training stage is required. DropBack differs from these because it substantially reduces the memory footprint and memory access counts *during* training as well as when the pruned network is used for inference.

Adding a term to the loss or additional parameters to encourage sparsity has also been examined in several papers [4, 49, 75, etc.]. While these pruning techniques are as effective as pruning using a retraining stage, they either massively increase the time to convergence, memory usage, or both. These pruning techniques require extra parameters for stochastic gradient descent to update, making the optimization task more difficult and increasing memory usage by several times. In contrast, DropBack is able to *reduce* the memory footprint of weights during training and converges at a rate equivalent to a non-pruning training algorithm. These alternative pruning techniques could be used to gradually reduce the computation and memory required while training a network, but no current work has done so, and additionally these existing techniques cannot provide an upper bound on memory usage from the very first iteration like DropBack.

In the next chapter, we evaluate the variants of DropBack on real-world deep neural networks, investigate the reason why DropBack is effective, and analyze the tradeoffs for both algorithms.

# Chapter 4

# Results and Discussion

DropBack was tested on three different datasets and a total of five different neural networks, achieving between 4× to 72× weight reduction without losing more than 2% in validation accuracy. On the more complicated networks, WRN-28-10 and Densenet, DropBack Algorithm 2 outperformed the current state-of-the-art pruning techniques on CIFAR-10 in accuracy and weight reduction. The DropBack variant that decays weights to zero, Algorithm 3, reduced the number of non-zero parameters by 98.6% on the WRN-28-10/CIFAR-10 with an acceptable accuracy loss of 1.4 percentage points. When the network architecture allows it, this scheme naturally results in zeroing out entire convolutional filters — with 94% of the 3x3 convolutional filters removed entirely — a further reduction in computation.

In the remainder of this chapter, we detail the methods, networks, and datasets used for testing, and discuss the results.

## 4.1 Methods

DropBackwas tested on five networks and three datasets, using standard industry tools. Results are directly comparable to other pruning papers.

### 4.1.1 Training Methods

DropBack was implemented using the Chainer deep neural network toolkit [63]; all runs, including DropBack runs, were trained on an NVIDIA 1080Ti GPU. We compared DropBack against four training regimes:

- a baseline implementation without any pruning;
- a straightforward magnitude-based pruning implementation where only the highest weights are kept after each iteration, i.e., Algorithm 1;
- variational dropout [32], which can progressively prune weights during training but at least doubles the memory footprint; and
- network slimming [42], a modern train-prune-retrain pruning method that achieves state-of-the-art results on modern network architectures.

All DropBack results were optimized using stochastic gradient descent without momentum, as all other optimization strategies cost significant extra memory. All methods without DropBack used the ADAM optimizer [31] with the parameters specified by the original papers.

### 4.1.2 Networks Used

A total of five deep neural networks are used to test DropBack. The first two are basic MLPs with a different numbers of neuron in their hidden layer, the third is a straightforward convolutional neural network, and the final two are state-of-the-art convolutional networks with complex, compact architectures. They are:

- LeNet-300-100 [39], a common benchmark MLP, with 786 neurons in the input layer, 300 neurons in the first hidden layer, 100 neurons in the second hidden layer, and 10 neurons in the output layer, for a total of 266K weights;
- MLP-100, a smaller variant of LeNet-300-100, with only 100 hidden neurons in the first hidden layer and no second hidden layer, for a total of 90K weights;
- VGG-S [70], shown in Figure 4.1, a smaller variant of VGG-16 [58] that omits one fully connected layer and reduces the other two to 512 neurons instead of 4,096 and adds modern batch normalization layers between its convolution blocks;
- Densenet [27], shown in Figure 4.2, a very densely interconnected variant of VGG aimed at improving training accuracy while reducing the weight count; and
- WRN-28-10 [71], a modern wide residual network with state-of-the-art performance on image tasks, which features a complex architecture with four

different different blocks, shown in Figure 4.3, and is discussed in more detail below.

LeNet-300-100 and the original VGG-16 are widely acknowledged to be very overprovisioned [30], with many more weights than necessary (i.e., later variants with fewer weights perform just as well or better), but are often used to demonstrate pruning effectiveness [14, 44, 46]. To avoid pruning purely due to the network being too large, we focus on the MLP-100 variant to LeNet-300-100, as well as a much smaller VGG-S variant of VGG-16 that reduces the weight count from 138M to 15M.

Densenet and WRN-28-10 are both much newer and more complex convolutional networks with fewer parameters, and were chosen precisely because their complex, more interconnected architectures make them very challenging to prune with existing techniques [41–43]. Densenet is constructed from stacks of convolutional layers, with the output of each stack being fed to *all* downstream stacks (rather than just the next layer as in a standard convolutional network). The authors of Densenet argue that this design removes the vanishing gradient problem, strengthens feature propagation, and reduces the number of parameters compared to standard convolutional neural networks. WRN-28-10 is constructed of four types of blocks, shown in Figure 4.3. Block type A has two stacked convolutions but also concatenates its inputs to the outputs of those convolutions, carrying the input feature maps downward without modification. Block type B is similar, but has two bottleneck layers — layers of 1×1 convolutions — surrounding a normal 3×3 convolutional layer. Block type C is the same as A, but with many more filters in each convolutional layer, while block type D adds a dropout layer [60] between pairs of convolutions. Wide residual networks are extremely effective at image classification tasks, but as the number of filters in the wide block are increased the parameter count dramatically increases.

### 4.1.3 Datasets

DropBack was evaluated on the MNIST digit recognition dataset [36], the Fashion-MNIST small grayscale image dataset [69], and the CIFAR-10 color image recognition dataset [34]. For MNIST, two networks were used: LeNet-300-100 and MLP-100. For Fashion-MNIST, we used VGG-S alone. For CIFAR-10, three networks were

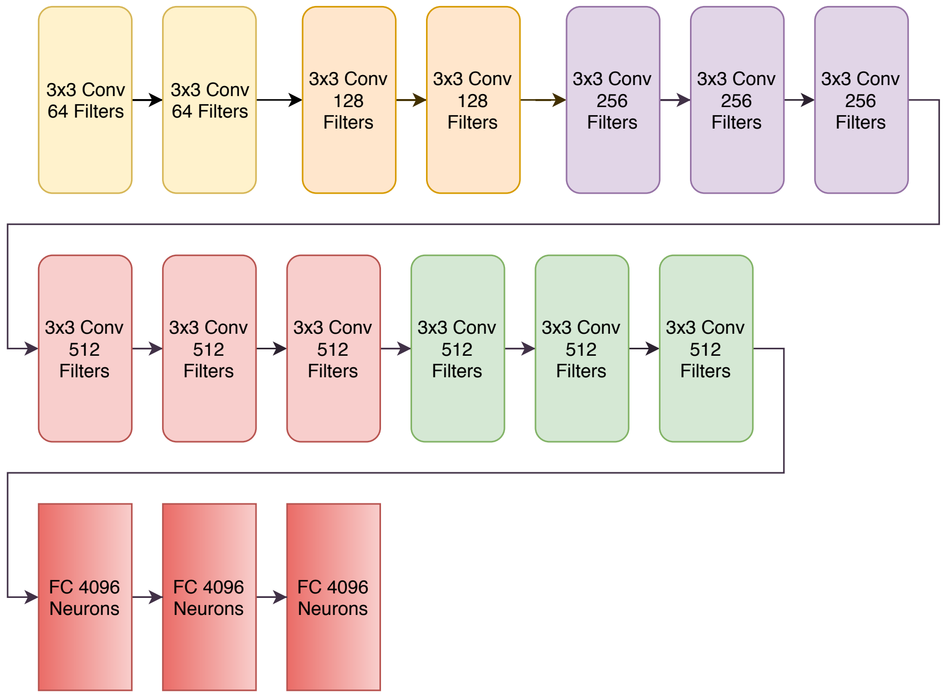


**Figure 4.1:** The VGG-16 network architecture. VGG-16 is an older convolutional neural network and is designed with a single path through the network, with different blocks of 3x3 filters stacked together.

used: VGG-S, Densenet [26], and WRN-28-10 [71].

## 4.2 Handwritten Digit Recognition (MNIST)

DropBack was first evaluated using Algorithm 1, Algorithm 2, and Algorithm 3 on the MNIST handwritten digit dataset using LeNet-300-100 MLP and MLP-100. Training was allowed for up to 100 epochs, and the initial learning rate of 0.4 was reduced every 25 epochs by a factor of 0.5. Training was stopped after five epochs in which accuracy did not improve over the previous best epoch.

### 4.2.1 Pruning by Weight Magnitude: Algorithm 1

To demonstrate that standard post-training pruning techniques cannot be used to prune networks at training time, we first examined Algorithm 1, a naive adaptation of

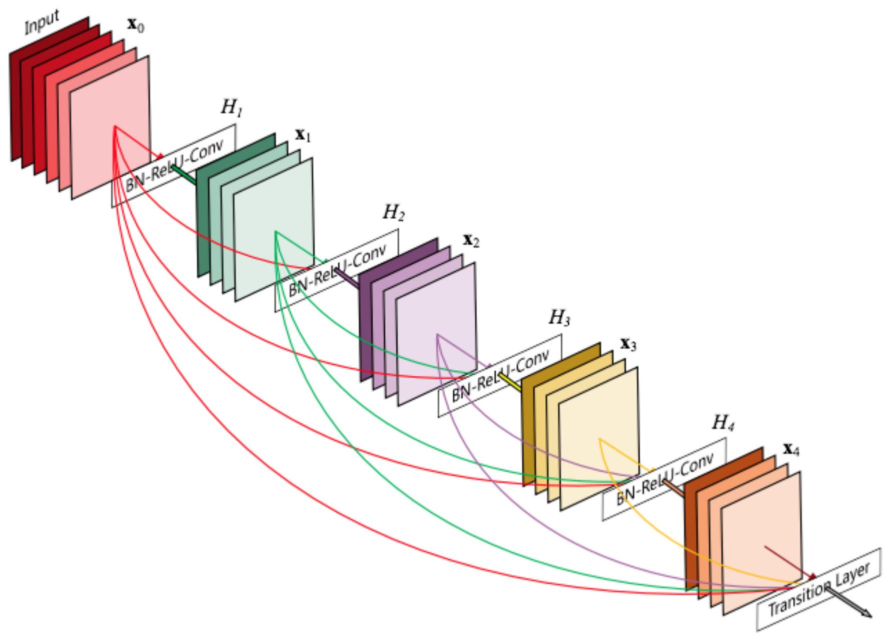


**Figure 4.2:** The Densenet network architecture. Densenet connects the output of each convolutional layer to the inputs of all downstream layers in order to propagate feature information more readily throughout the network. Figure taken from [27].

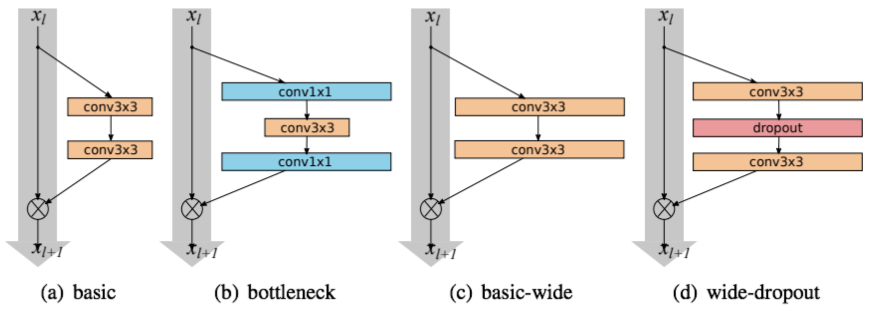


**Figure 4.3:** The types of wide residual network blocks. wide residual networks are built from these four blocks. Figure taken from [71].

| LeNet-300-100 | Val. Error | Weight Reduction | Best Epoch | Freeze Epoch |
|---|---|---|---|---|
| Baseline 266K | 1.41% | 1× | 65 | N/A |
| Naive 50K | 86.56% | 5.33× | 94 | 0 |
| Naive 20K | 91.1% | 13.3× | 67 | 0 |
| Naive 5K | 89.89% | 53.32× | 45 | 0 |
| Naive 1.5K | 90.0% | 177.74× | 54 | 0 |
| **MLP-100** | **Val. Error** | **Weight Reduction** | **Best Epoch** | **Freeze Epoch** |
| Baseline 90K | 1.70% | 1× | 47 | N/A |
| Naive 50K | 11.56% | 1.8× | 31 | 0 |
| Naive 20K | 88.9% | 4.5× | 87 | 0 |
| Naive 5K | 89.1% | 18× | 65 | 0 |
| Naive 1.5K | 89.5% | 60× | 56 | 0 |

**Table 4.1:** LeNet-300-100 (top) and MLP-100 (bottom) on MNIST, using the naive prune-by-weight-magnitude Algorithm 1. Naive 50K refers to a configuration where 50,000 weights are retained during training, Naive 5K to a configuration with 5,000 retained weights, and so on.

post-training pruning (see Section 3.1). In this scheme, only the highest-magnitude weights are retained during every iteration, and the remaining weights are set to zero.

Table 4.1 shows the results for the baseline (unpruned) model and four configurations of this naive scheme using Algorithm 1. Neither LeNet-300-100 or MLP-100 can be effectively pruned using this scheme, with a 5× weight reduction ratio resulting in an error only slightly above random (since there are 10 digits, a random guess will be correct 10% of the time). MLP-100 can be reduced by 1.8× at the cost of a substantially lower accuracy, unfortunately, even higher weight reductions result in no learning, with network output remaining random.

Because Algorithm 1 fails even on the simplest dataset used in this thesis, we omit its results from the remainder of this chapter.

### 4.2.2 Pruning by Accumulated Gradient Magnitude: Algorithm 2

We next examined Algorithm 2, which retains the highest accumulated gradients and restores the remaining weights to their initialization-time values (see Section 3.2).

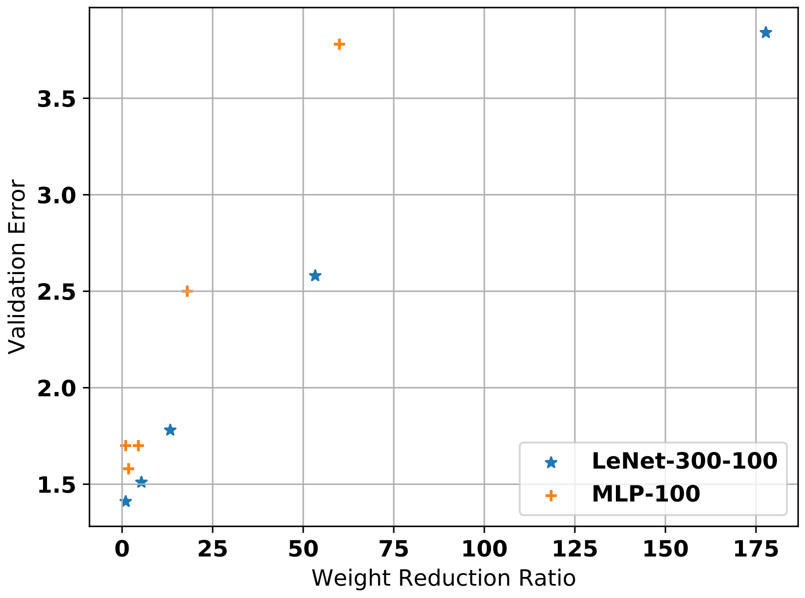


**Figure 4.4:** The accuracy vs. weight reduction trade-off for LeNet-300-100 and MLP-100 on Algorithm 2.

Table 4.2 shows the results on the MNIST digit dataset for the baseline (unpruned) model and four configurations of DropBack, retaining 50,000 weights (1.8× weight reduction), 20,000 weights (4.5× weight reduction), 5,000 weights, and 1,500 weights (60× weight reduction), respectively. Figure 4.4 shows the same data visually.

### Accuracy and Weight Reduction

Using MLP-100 with a modest 1.8× reduction in weights, DropBack slightly exceeds the accuracy of the baseline model. This slight accuracy improvement matches the trend reported in prior work such as DSD [20], where a sparse layer that omits 30% to 50% weights outperforms the baseline dense model with all of the weights retained. However, DSD first trains the network to convergence on the complete parameter set, and only then prunes some weights and retrains the resulting sparse network, with this process repeated several times (the final network output of DSD is not sparse, and therefore not included in this thesis as comparison). In contrast,

| LeNet-300-100 | Val. Error | Weight Reduction | Best Epoch | Freeze Epoch |
|---|---|---|---|---|
| Baseline 266K | 1.41% | 1× | 65 | N/A |
| DropBack 50K | 1.51% | 5.33× | 24 | 10 |
| DropBack 20K | 1.78% | 13.3× | 33 | 20 |
| DropBack 5K | 2.58% | 53.32× | 32 | 20 |
| DropBack 1.5K | 3.84% | 177.74× | 97 | 40 |
| **MLP-100** | **Val. Error** | **Weight Reduction** | **Best Epoch** | **Freeze Epoch** |
| Baseline 90K | 1.70% | 1× | 47 | N/A |
| DropBack 50K | 1.58% | 1.8× | 24 | 5 |
| DropBack 20K | 1.70% | 4.5× | 32 | 5 |
| DropBack 5K | 2.50% | 18× | 27 | 20 |
| DropBack 1.5K | 3.78% | 60× | 26 | 20 |

**Table 4.2:** LeNet-300-100 (top) and MLP-100 on the MNIST digit dataset, using Algorithm 2.

DropBack achieves this accuracy improvement *without* ever storing the full dense configuration of the network, and without repeatedly retraining the network to convergence.

The larger LeNet-300-100 MLP sees a slight drop in accuracy, at 50K retained weights, but the initial weight reduction of 5.33× is much higher than MLP-100. It also reaches maximum accuracy nearly 3× fewer epochs than Baseline. Further reducing the model to 20K weights (4.5× weight reduction) results in nearly the same accuracy as the baseline, and convergence in about 50% fewer epochs than Baseline.

With an extreme weight reduction of 1.5K tracked weights, the error rate roughly doubles. However, the nearly 60× reduction in the weight storage requirement for MLP-100 and 177× for LeNet-300-100 potentially offers an attractive design point for low-power embedded accelerators in future mobile and edge devices.

### Freezing Epoch

We also investigated the effect of freezing the tracked gradient selection process (see Section 3.2) early during training. The freeze epoch was selected by a hyper-parameter sweep.

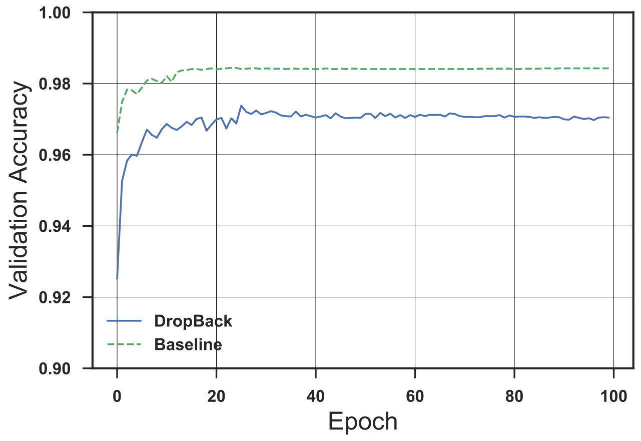


**Figure 4.5:** Rate of convergence using Algorithm 2 for LeNet-300-100 for our technique our and the baseline model. Note that the y-axis starts as 0.90, and the final accuracies are within 1% of each other.

For both MNIST networks, freezing sooner to reduce the computational overhead results in lower achieved accuracy at very high weight reduction ratios, but for smaller weight reduction ratios freezing early has little effect on the overall accuracy. When the weight reduction ratio is small, the weights being swapped in and out of the tracked set are considerably less important than in high weight reduction ratios, therefore freezing early is less likely to choose a poor tracked parameter set. Figure 4.5 shows the rate of convergence for DropBack and the baseline on the LeNet-300-100 network; despite converging to the nearly same accuracy DropBack has more noise than the Baseline.

### Allocation of Retained Weights

Next, we examined which network layers retained the most weights. We reasoned that as the weight reduction factor increases and accuracy drops, the network will reduce the precision of feature detection and put more resources into decision-making.

Table 4.3 shows that the number of parameters retained per layer indeed varies

| layer | DropBack 1.5K | DropBack 10K | Baseline |
|---|---|---|---|
| fc1 (100×784) | 734 (48.9%) | 7223 (72.2%) | 78500 (87.6%) |
| fc2 (100×100) | 512 (34.1%) | 2128 (21.3%) | 10100 (11.3%) |
| fc3 (100×10) | 254 (16.9%) | 549 (5.5%) | 1010 (1.1%) |
| Total | 1500 | 10000 | 89610 |

**Table 4.3:** Number of gradients for each layer retained in the final trained MLP-100 network.

| LeNet-300-100 | Val. Error Alg. 2 | Val. Error Alg. 3 | Best Epoch |
|---|---|---|---|
| Baseline 266K | 1.41% | 1.41% | 65 |
| DropBack 50K | 1.51% | 1.51% | 43 |
| DropBack 20K | 1.78% | 2.10% | 50 |
| DropBack 5K | 2.58% | 3.32% | 58 |
| DropBack 1.5K | 3.84% | 10.70% | 81 |
| **MLP-100** | **Val. Error Alg. 2** | **Val. Error Alg. 3** | **Best Epoch** |
| Baseline 90K | 1.70% | 1.70% | 47 |
| DropBack 50K | 1.58% | 1.69% | 44 |
| DropBack 20K | 1.70% | 2.06% | 41 |
| DropBack 5K | 2.50% | 2.06% | 70 |
| DropBack 1.5K | 3.78% | 17.14% | 78 |

**Table 4.4:** The MNIST digit dataset using LeNet-300-100 (top) and MNIST-100-100, using Algorithm 3. All models where frozen from epoch 25.

depending on the number of tracked weights. The smaller DropBack 1.5K network allocates a much higher proportion of its weights to the later layers compared to the DropBack 10K network and the baseline. As later layers make decisions based on the features provided by earlier layers, this result suggests that the more reduced network indeed assigns proportionally more neurons to the critical decision-making tasks.

### 4.2.3 DropBack with Weight Decay: Algorithm 3

Finally, we investigated whether decaying the untracked weights to zero (as opposed to recomputing their initialization-time values) was effective on these small networks.

We examined Algorithm 3 on the MNIST digit dataset, using the same networks as with Algorithm 2 above and an untracked weight value decay rate $\delta = 0.90$. This adds an extra floating point multiply to every parameter in the network, however the cost of doing so is low since the number of weights in all networks is in the millions, and the number of operations is in the billions. Additionally, this extra cost is only present for the first 1000 iterations before all initial parameters have decayed to zero.

The results are shown in Table 4.4 and Figure 4.4. Overall, compared with the results from Algorithm 2 results in Table 4.2, decaying weights resulted in increase validation error rates for a given weight reduction ratio, especially on the highest level of weight reduction ratios tested. For example, with 1.5K weights retained, the error increases to 10.70% on LeNet-300-100 and 17.14%; With the exception of LeNet-300-100 DropBack 50K, all of the other weight reduction ratios also see an increase in error compared to Algorithm 2. This appears to be because these relatively small networks rely on the scaffolding provided by the initial parameters, and without this the remaining parameters have trouble retaining generalization accuracy.

At the higher end of the weight reduction ratios, the error is too high to be usable. DropBack 20K offers reasonable error for weight reduction, LeNet-300-100 increases from 1.78% error to 2.10% with a weight reduction ratio of 13.3×, and MLP-100 increases from 1.70% to 2.06% for a weight reduction ratio of 4.5×.

## 4.3 Fashion Item Icon Recognition (Fashion-MNIST)

We next evaluated DropBack on the VGG-S convolutional network. Because the MNIST dataset is not challenging for deep convolutional networks like VGG, which can achieve 98% accuracy [58], we evaluated it on Fashion-MNIST, a dataset that is the same size as MNIST (28×28 grayscale images, 10 categories) but is also much more challenging for modern networks in terms of achieved accuracy [69]. Stochastic gradient descent without momentum was used with a maximum of 200 epochs, with the learning rate constant for all epochs; no data augmentation was performed.

Both Algorithm 2 and Algorithm 3 were evaluated, for a variety of tracking constraints and freezing epochs in order to explore the weight reduction vs. accuracy trade-offs. The results are presented in Table 4.5 for Algorithm 2 and in Table 4.6 for Algorithm 3.

### 4.3.1 Pruning by Accumulated Gradient Magnitude: Algorithm 2

#### Accuracy and Weight Reduction

DropBack Algorithm 2 achieved slightly higher accuracy than the Baseline while reducing the model 5×, resulting in a network where only 3M parameters out of the approximately 15M original parameters differed from their initial values. A higher weight reduction ratio of nearly 30× increased the error to 7.4%, likely an acceptable tradeoff for embedded edge devices with very stringent weight storage budgets.

Compared with results shown later on the CIFAR-10 dataset (see Section 4.4 and Table 4.7), the network can be reduced far more (up to ~ 30×) with a lower error rate that what is observed for CIFAR-10. This is because Fashion-MNIST is a simpler dataset than CIFAR-10: it consists of 28×28×1 grayscale images, versus 32×32×3 images in full color. Since the images are grayscale, the network does not need to distinguish different features in each color channel, and fewer convolutional filters overall are required.

#### Freezing Epoch

Like the earlier experiments on MNIST, we also examined the effect of selecting the freezing epoch on accuracy. Unlike in the case of MNIST, where higher reduction ratios resulted in a later optimal freezing epoch, the optimal freezing epoch for the MNIST-Fashion experiments did not appear to depend on the target reduction ratio. The optimal freezing ratio was chosen in the same way as with the MNIST experiments, by hyper-parameter sweep. In all but the DropBack 5M case, the optimal point was 30 epochs. Possibly, the parameter selection on Fashion-MNIST is easier due to the makeup of the dataset compared to the standard MNIST dataset.

### 4.3.2 DropBack with Weight Decay: Algorithm 3

Finally, we investigated the weight-decay variant of DropBack on the MNIST-Fashion task using the VGG-S network. The results are shown in Table 4.6 and Figure 4.6.

| Fashion MNIST | Validation Error | Weight Reduction | Best Epoch | Freeze Epoch |
|---|---|---|---|---|
| VGG-S Baseline 15M | 5.43% | 1× | 178 | N/A |
| VGG-S DropBack 5M | 5.14% | 3.0× | 105 | 25 |
| VGG-S DropBack 3M | 5.43% | 5.0× | 151 | 30 |
| VGG-S DropBack 500K | 7.39% | 30.0× | 175 | 30 |
| VGG-S DropBack 200K | 13.44% | 75.0× | 189 | 30 |
| VGG-S DropBack 100K | 19.80% | 150.0× | 172 | 30 |

**Table 4.5:** VGG-S on Fashion-MNIST, using Algorithm 2.

| Fashion MNIST | Val. Error Alg. 2 | Val. Error Alg. 3 | Best Epoch |
|---|---|---|---|
| VGG-S Baseline 15M | 5.43% | 5.43% | 178 |
| VGG-S DropBack 5M | 5.14% | 5.32% | 75 |
| VGG-S DropBack 3M | 5.43% | 5.62% | 158 |
| VGG-S DropBack 500K | 7.39% | 6.50% | 192 |
| VGG-S DropBack 200K | 13.44% | 8.87% | 93 |
| VGG-S DropBack 100K | 19.80% | 16.34% | 40 |

**Table 4.6:** VGG-S on Fashion-MNIST, using Algorithm 3.

At the lower weight reduction ratios, Algorithm 3 results in minimal increases in error compared to Algorithm 2 at the lower weight reduction ratios, but in exchange creates useful levels of computational sparsity by zeroing a large proportion of weights. For example, DropBack 3M has 80% of the model parameters decayed to zero by the end of training. During training, this sparsity does not result in computation savings because the weight decay is achieved over time; however, at inference time, many of the weights are zero and computation savings can be significant.

In the higher range of weight reduction ratios, on the other hand, Algorithm 3 actually *outperforms* Algorithm 2 by 1–3%. At 500K tracked parameters, only 3.3% of the network is non-zero. Using a coordinate-list sparse matrix representation, the sparsified weight matrix of VGG-S trained by DropBack with pruning to 500K weights would use only 3.77MB compared to 57MB for the unpruned baseline in a dense matrix representation, including metadata overhead. Compared to Algorithm 2, storing the weights as a standard sparse matrix *without* the requirement to regenerate

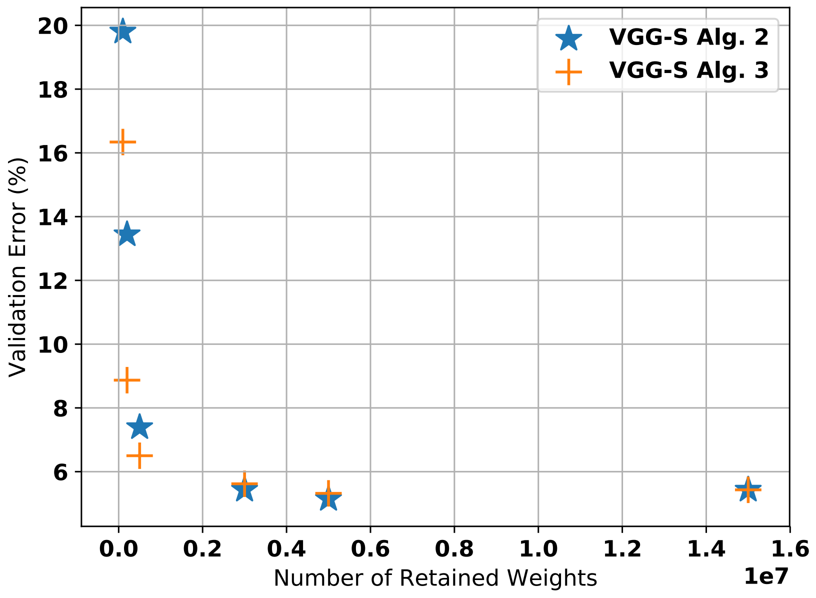


**Figure 4.6:** Weight reduction ratio vs. accuracy for VGG-S on Fashion MNIST.

their initial values is beneficial for further reducing inference time energy, as weights no longer need to be recomputed using the XORSHIFT RNG. An accelerator such as the Efficient Inference Engine [19] can, according to the authors, leverage this level of sparsity to improve energy efficiency by 10× compared to a dense weight matrix representation.

## 4.4 Color Image Classification (CIFAR-10)

Finally, we evaluated DropBack on the CIFAR-10 dataset. This dataset presents a much more challenging task than MNIST or Fashion-MNIST because (a) the images are larger (32×32 pixels vs. 28×28), and, more importantly, (b) the images are in color, which means that the network must learn to reason about and combine features detected in three separate color channels.

In addition to the fairly straightforward VGG-S network, we also evaluated on Densenet and WRN-28-10. These are modern networks with complex architectures and state-of-the-art performance on image classification tasks. We specifically

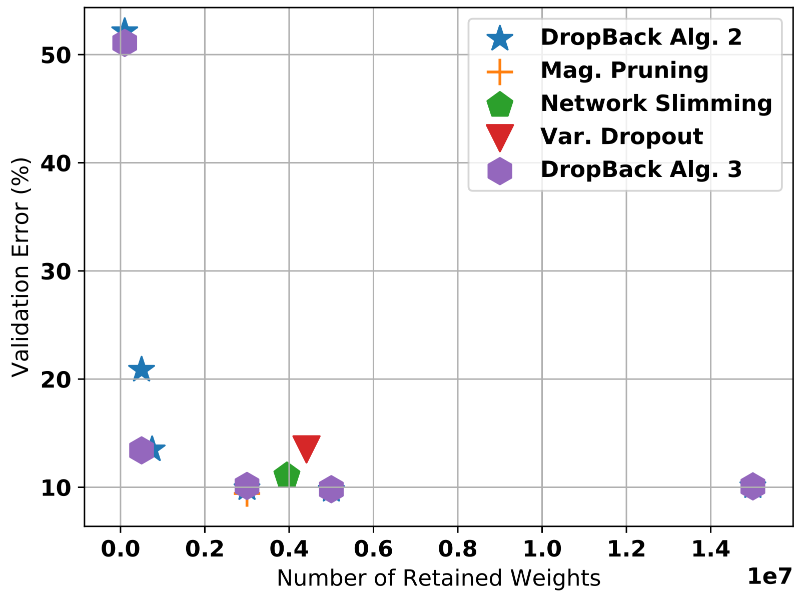


**Figure 4.7:** Weight reduction ratio vs. accuracy for VGG-S, using Algorithm 2 and Algorithm 3.

selected them because both networks are compact with very dense connections, and are very challenging for prior pruning methods to optimize [42]. VGG-S was trained for 300 epochs, while Densenet and WRN-28-10 were trained for 500 epochs; the highest-accuracy epoch was selected for the final result. For all three experiments, the learning rate started at 0.4 and decayed 0.5× every 25 epochs; no data augmentation was performed.

### 4.4.1 Pruning by Accumulated Gradient Magnitude: Algorithm 2

#### Accuracy and Weight Reduction

Table 4.7 and figures 4.7, 4.8, and 4.9 show how DropBack compares to variational dropout, network slimming, and magnitude-based pruning on VGG-S, Densenet, and WRN-28-10. Overall, DropBack can achieve comparable (or even slightly improved) accuracy on VGG-S and Densenet with a five-fold weight reduction, and

| CIFAR-10 | Validation error | Weight reduction | Best epoch | Freeze epoch |
|---|---|---|---|---|
| VGG-S Baseline 15M | 10.08% | 1× | 214 | N/A |
| VGG-S DropBack 5M | 9.75% | 3× | 127 | 5 |
| VGG-S DropBack 3M | 9.90% | 5× | 128 | 20 |
| VGG-S DropBack 750K | 13.49% | 20× | 269 | 35 |
| VGG-S DropBack 500K | 20.85% | 30× | 201 | 15 |
| VGG-S DropBack 100K | 52.15% | 150× | 30 | 15 |
| VGG-S Var. Dropout | 13.50% | 3.4× | 200 | N/A |
| VGG-S Mag Pruning .80 | 9.42% | 5.0× | 182 | N/A |
| VGG-S Slimming | 11.08% | 3.8× | 196 | N/A |
| Densenet Baseline 2.7M | 6.48% | 1× | 382 | N/A |
| Densenet DropBack 500K | 5.86% | 4.5× | 409 | N/A |
| Densenet DropBack 100K | 9.42% | 27× | 307 | N/A |
| Densenet Var. Dropout | 90% | 1× | N/A | N/A |
| Densenet Mag Pruning .75 | 6.41% | 4.0× | 480 | N/A |
| Densenet Slimming | 5.65% | 2.9× | 495 | N/A |
| WRN-28-10 Baseline 36M | 3.75% | 1× | 326 | N/A |
| WRN-28-10 DropBack 8M | 3.85% | 4.5× | 384 | N/A |
| WRN-28-10 DropBack 7M | 4.02% | 5.2× | 417 | N/A |
| WRN-28-10 DropBack 5M | 4.20% | 7.3× | 304 | N/A |
| WRN-28-10 DropBack 1M | 31.18% | 36.48× | 460 | N/A |
| WRN-28-10 DropBack 500K | 41.02% | 72.96× | 494 | N/A |
| WRN-28-10 DropBack 100K | 83.12% | 364.80× | 57 | N/A |
| WRN-28-10 Var. Dropout | 90% | 1× | N/A | N/A |
| WRN-28-10 Mag Pruning .75 | 26.52% | 4× | 109 | N/A |
| WRN-28-10 Slimming .75 | 16.640% | 4× | 173 | N/A |

**Table 4.7:** Validation accuracy and weight reduction ratio of several pruned networks on the CIFAR-10 dataset, trained using Algorithm 2.

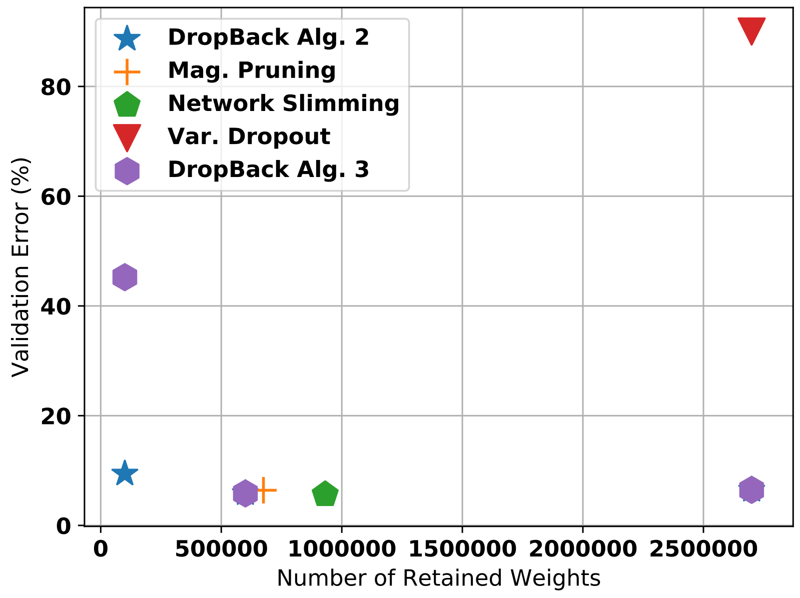


**Figure 4.8:** Weight reduction ratio vs. accuracy for Densenet, using Algorithm 2 and Algorithm 3.

up to 20×–30× weight reduction if some accuracy is sacrificed. On WRN-28-10, DropBack achieves 7× weight reduction with less than 0.5% accuracy compared to other techniques which all offer at best 4× weight reduction at over 10% increase in error rate.

WRN-28-10 and Densenet are challenging, as they are already quite dense for the accuracy level they achieve. Variational dropout works well only on VGG-S and fails to converge — that is, fails to achieve better-than-random accuracy — on Densenet and WRN. Magnitude-based pruning tops out at a worse accuracy than DropBack on WRN-28-10 and Densenet, despite offering less weight reduction when pruning 80% and 75% of the parameters, respectively. Finally, network slimming achieves slightly better top accuracy on Densenet at the cost of 36% less weight reduction, but results in dramatic accuracy loss when applied to WRN-28-10; this is in line with recent work which has shown that WRN is hard to reduce more than about 2× without losing significant accuracy [41–43]. Only DropBack is able to achieve

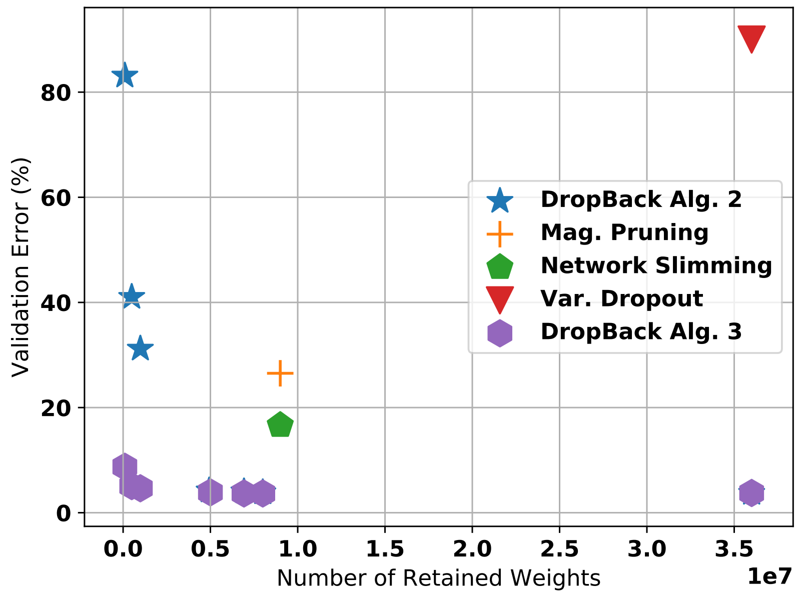


**Figure 4.9:** Weight reduction ratio vs. accuracy for WRN-28-10, using Algorithm 2 and Algorithm 3.

weight reduction on the order of 5× with little to no accuracy loss on WRN-28-10 and Densenet.

## Convergence and Freezing

With the more complicated networks WRN-28-10 and Densenet, freezing early resulted in significant accuracy drops of ~10% compared to not freezing. This is because these networks are much more difficult to train than VGG-S, as demonstrated by the larger best epoch values in Table 4.7. Intuitively, with higher weight reduction ratios and more complex networks, freezing early results in a more substantial accuracy drop because parameters that are on the edge of being included in the tracked set are far more critical. With larger tracked sets, weights that are being swapped in and out of the tracked set have smaller gradients, therefore contribute less to the error. With more complex networks, additional noise, such as the noise added by parameter selection changing each iteration, can be beneficial to training [50].

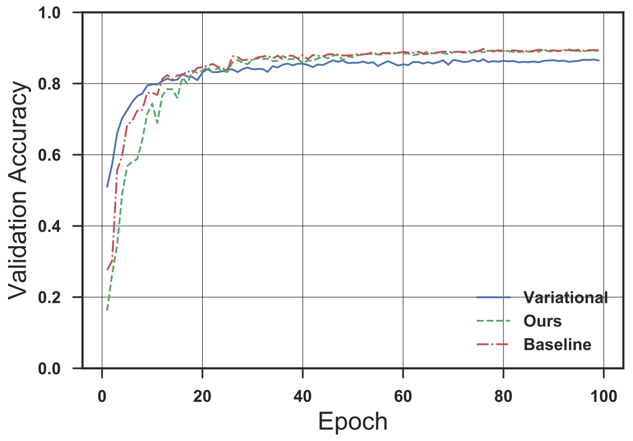


**Figure 4.10:** VGG-S CIFAR-10 epoch vs. validation accuracy for our method at 5M tracked parameters, variational dropout, and the baseline model.

Therefore, WRN-28-10 and Densenet were run without freezing.

Figure 4.10 shows that DropBack initially learns slightly more slowly than the uncompressed baseline on VGG-S, but exhibits the same convergence behavior as the baseline after about 20 epochs. Variational dropout, in contrast, learns more quickly initially but converges to a lower accuracy, essentially because the rapid movement results in numerical instability and poor performance [23] (see Section 4.5 for details).

### 4.4.2 DropBack with Weight Decay: Algorithm 3

#### Accuracy and Weight Reduction

The results of training networks on CIFAR-10 using Algorithm 3 are shown in Table 4.8. In almost every case, Algorithm 3 improves the weight reduction ratio.

The level of weight reduction achieved for an acceptable accuracy loss depends on the network used. The newer WRN-28-10 network is reduced the most, and at high compression levels is still more accurate than any VGG-S result. Densenet

| CIFAR-10 | Validation error | Best epoch |
|---|---|---|
| VGG-S Baseline 15M | 10.08% | 214 |
| VGG-S DropBack 5M | 9.82% | 234 |
| VGG-S DropBack 3M | 10.12% | 188 |
| VGG-S DropBack 500K | 13.41% | 227 |
| VGG-S DropBack 100K | 51.08% | 255 |
| Densenet Baseline 2.7M | 6.48% | 382 |
| Densenet DropBack 500K | 5.83% | 479 |
| Densenet DropBack 100K | 45.24% | 437 |
| WRN-28-10 Baseline 36M | 3.75% | 326 |
| WRN-28-10 DropBack 8M | 3.66% | 493 |
| WRN-28-10 DropBack 7M | 3.66% | 493 |
| WRN-28-10 DropBack 5M | 3.87% | 478 |
| WRN-28-10 DropBack 1M | 4.65% | 460 |
| WRN-28-10 DropBack 500K | 5.06% | 562 |
| WRN-28-10 DropBack 100K | 8.74% | 362 |

**Table 4.8:** Validation accuracy and weight reduction ratio of several pruned networks on CIFAR-10 using Algorithm 3

has far fewer parameters than the baseline WRN-28-10 or VGG-S networks, but Algorithm 3 is still able to achieve weight reduction ratio while maintaining accuracy. Compared to VGG-S, the advantages of the newer, more complex architectures are clear especially when reduced to 500K and 100K tracked parameters. Densenet at 500K tracked parameters gains almost 1% in test accuracy, whereas VGG-S loses 3.4%, compared to the baseline accuracy result for the respective network.

Using DropBack with 100K tracked weights, WRN-28-10 has the highest weight reduction ratio and significantly lower validation error than VGG-S or Densenet; Densenet, on the other hand, suffers a colossal increase in error when only 100K weights are tracked. This is likely due to its unique highly interconnected architecture: in a standard CNN like VGG-S, the outputs of one layer are fed into only the next layer. In Densenet the output of each layer is fed to into every layer below. This causes each weight in Densenet to affect more of the output than either WRN-28-10 or VGG-S

The more sparsely connected WRN-28-10 network retains much higher accuracy even with high weight reduction ratios. At 100K, the error only doubles compared to the baseline, and WRN-28-10 still outperforms the best (lowest error) version of VGG-S.

These results suggest that, (a) that network architecture has a significant impact on the possible weight reduction with DropBack, and (b) that sparsely connected architectures tend to perform better.

### Filter-Level Sparsity

Finally, we asked whether the weights that were pruned away correlated with the convolutional filter structure of the network. Because each filter represents a different feature, we reasoned that most of the weight pruning would occur in units of entire filters.

To investigate filter-level sparsity, we asked how many weights remained (i.e., had non-zero values) in each 3×3 convolutional filter at the end of the training process using Algorithm 3. We selected Weight reduction levels of 500K, and 100K tracked parameters as these both still outperformed the VGG-S baseline. The histograms show filters with 0 to 9 weights for each network and training configuration in Figures 4.11to 4.18, inclusive.

For WRN-28-10, the vast majority of 3×3 filters have been decayed to zero completely, allowing them to be removed from the network entirely. This is a significant computational savings, as each filter requires approximately 20 operations per pixel. Figure 4.11 shows the number of non-zero weights per 3×3 convolution for DropBack 500K, while Figure 4.13 shows the same for the case of 100K tracked parameters.

We also investigated the filters that were not entirely removed and asked how many filter values are non-zero. Figures 4.12 and 4.14 show the same histograms as above but now without the all-zero filters, again for 500K and 100K, respectively. In both cases, the vast majority of the remaining filters have only a single value. This again results in substantial reduction in the computation effort, as applying a single-value filter requires just a single operation per pixel.

In contrast to WRN-28-10, we observed fewer filters entirely reduced to zero in VGG-S and Densenet, with either 100K or 500K tracked weights. On VGG-S with 500K tracked parameters, only 88% of the filters were set completely to zero compared with the 94% of the filters from WRN-28-10 500K. Figure 4.15 and Figure 4.16 show how dense the VGG-S filters are overall compared to the WRN-28-10 filters in Figure 4.11 and Figure 4.12. Like WRN-28-10, VGG-S had many single value filters (Figure 4.16), but had relatively more high complexity filters whereas WRN-28-10 had essentially no filters with more than four non-zero weights. This suggests that VGG-S requires more complex filters than the WRN-28-10 network, perhaps because the more complex units that comprise the WRN-28-10 architecture pre-encode some of the necessary computation and allow the filters themselves to be less complex. VGG-S also could not be reduced as far as WRN-28-10 without catastrophic accuracy loss.

Densenet, on the other hand, relies on very complex filters, and did not achieve as much filter-level sparsity as VGG-S or WRN-28-10. The histograms are shown in Figures 4.17 and 4.18. While 64% of the filters are entirely zero, the second highest category of filter were filters with 7 non-zero values.

This result is intuitive, as the Densenet network used in this thesis has 5× fewer parameters than the next smallest network (VGG-S), but no complex building blocks like WRN-28-10 to account for some of the complexity. Therefore, to achieve better or similar accuracy as VGG-S network, the filters will have to incorporate more information. This more dense representation of information also helps explain Densenet's poor performance at higher weight reduction ratios — while some filters can be simplified, there are fewer filters overall to modify. Removing a single value from a filter in Densenet has far more impact than on VGG-S or WRN-28-10.

### 4.4.3 Weight Reduction: Algorithm 2 vs. Algorithm 3

The question of selecting DropBack Algorithm 2 versus DropBack Algorithm 3 depends greatly on the network, weight reduction desired, and energy reduction required. Table 4.9 shows a side-by-side comparison of Algorithms 2 and 3, using results from Section 4.4.

VGG-S performs better at higher weight reduction levels when using Algorithm 3, with the added benefit of the computational sparsity from the decayed to 0 initial

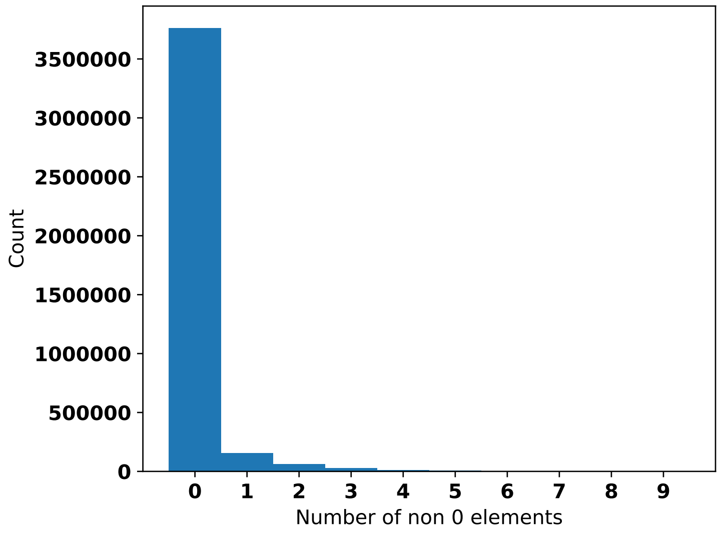


**Figure 4.11:** Histogram of 3x3 convolutional filters in WRN-28-10 with DropBack 500K Algorithm 3, including filters with all zeros.

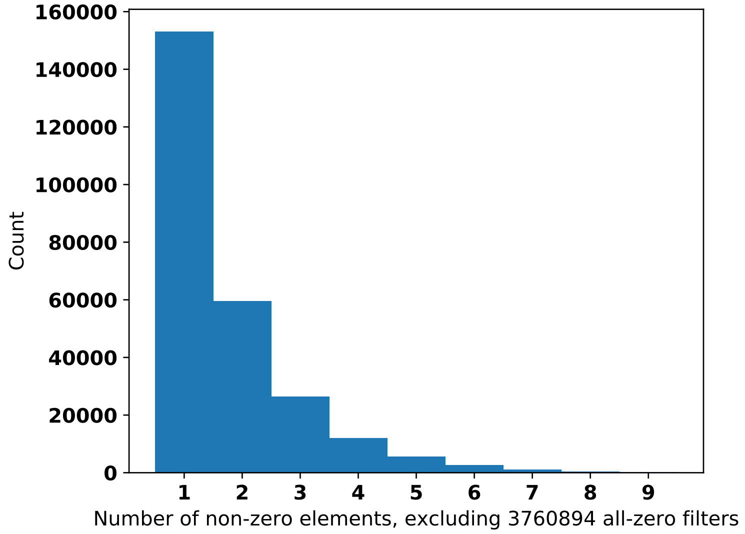


**Figure 4.12:** Histogram of 3x3 convolutional filters in WRN-28-10 with DropBack 500K Algorithm 3, excluding filters with all zeros.

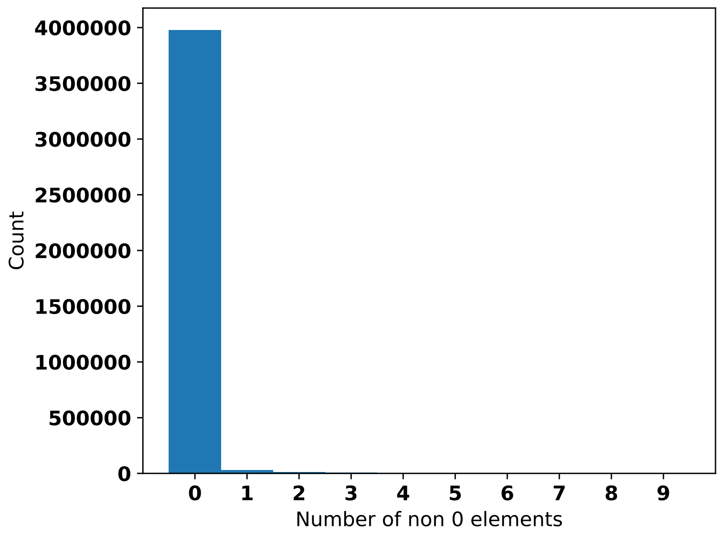


**Figure 4.13:** Histogram of 3x3 convolutional filters in WRN-28-10 with Drop-Back 100K Algorithm 3, including filters with all zeros.

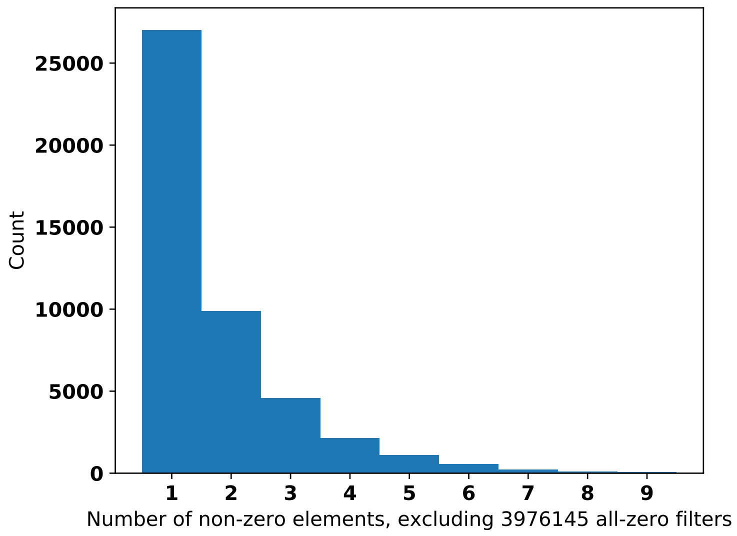


**Figure 4.14:** Histogram of 3x3 convolutional filters in WRN-28-10 with Drop-Back 100K Algorithm 3, excluding filters with all zeros.

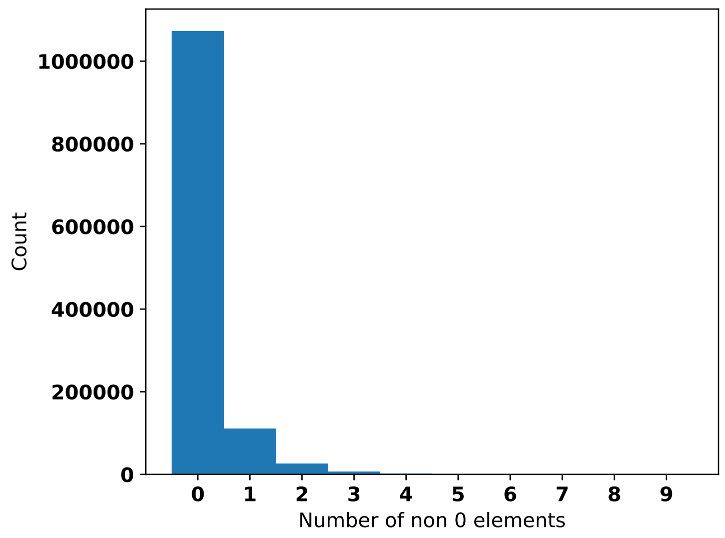


**Figure 4.15:** Histogram of 3x3 convolutional filters in VGG-S with DropBack 500K Algorithm 3, including filters with all zeros.

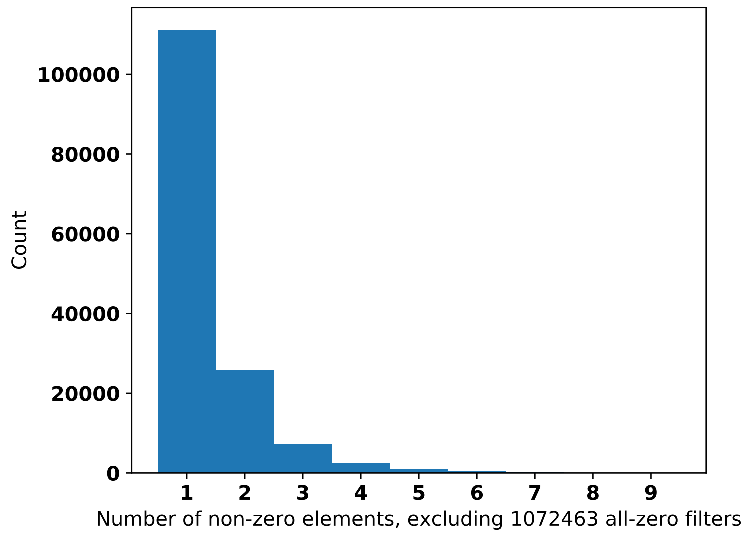


**Figure 4.16:** Histogram of 3x3 convolutional filters in VGG-S with DropBack 500K Algorithm 3, excluding filters with all zeros.

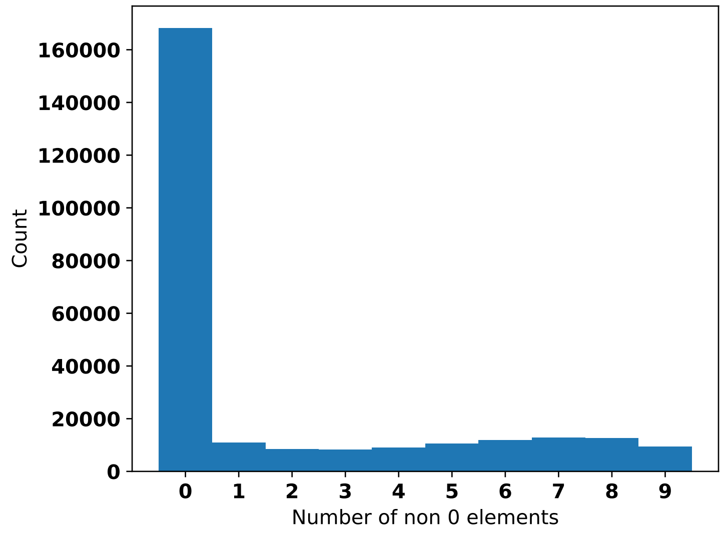


**Figure 4.17:** Histogram of 3x3 convolutional filters in Densenet with DropBack 500K Algorithm 3, including filters with all zeros.

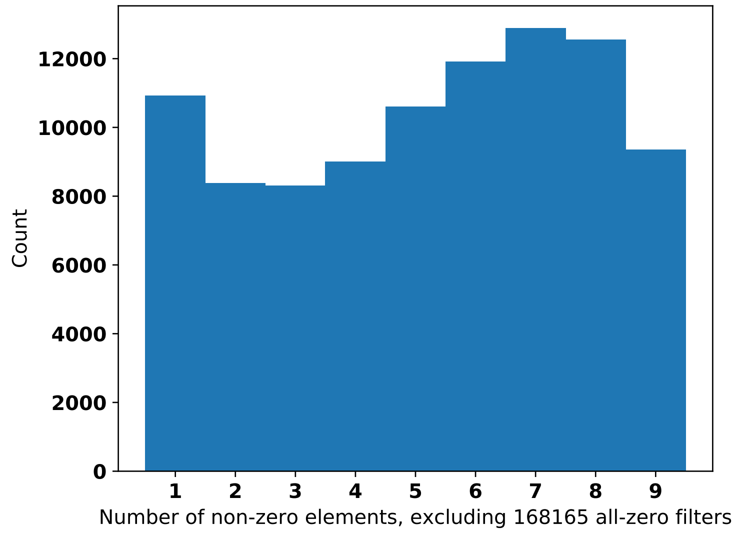


**Figure 4.18:** Histogram of 3x3 convolutional filters in Densenet with DropBack 500K Algorithm 3, excluding filters with all zeros.

| CIFAR-10 | Alg. 2 | Alg. 3 |
|---|---|---|
| VGG-S DropBack 5M | 9.75 % | 9.82% |
| VGG-S DropBack 500K | 20.85 % | 13.41% |
| VGG-S DropBack 100K | 90.00 % | 51.08% |
| Densenet DropBack 500K | 5.86 % | 5.83% |
| Densenet DropBack 100K | 9.42 % | 45.24% |
| WRN-28-10 DropBack 8M | 3.85 % | 3.66% |
| WRN-28-10 DropBack 500K | 59.89 % | 5.06% |
| WRN-28-10 DropBack 100K | 89.86 % | 8.74% |

**Table 4.9:** Comparing Algorithm 2 and Algorithm 3 on CIFAR-10.

values. At less extreme amounts of weight reduction, DropBack 5M, Algorithm 2 is very slightly better, although the computational sparsity obtained by Algorithm 3 makes the 0.07% increase in error acceptable.

Densenet, on the other hand, exhibits very different behavior when pruned. At low weight reduction levels, both algorithms perform the same, with Algorithm 3 taking longer to converge while introducing 78% weight sparsity. The weight reduction factor at DropBack 100K for Densenet is less than what WRN-28-10 and VGG-S achieved, but Algorithm 3 suffers a significant increase in validation error. This is because the densely connected architecture of the network causes too many activations to become zero, removing substantial expressive power of the already compact network. Leaving the initial parameters intact with Algorithm 2 results, on the other hand, preserves more accuracy and outperforms the baseline VGG-S network.

Finally, on WRN-28-10, Algorithm 3 outperforms Algorithm 2 at all levels of weight reduction, and does not lose accuracy as quickly at high levels of weight reduction. Using DropBack 100K Algorithm 3, WRN-28-10 outperforms the best VGG-S network even with the largest weight reduction ratio of 364.80×. While WRN-28-10 is the largest network used for CIFAR-10, and thus should see the highest weight reduction ratios, no other network was able to achieve a comparable accuracy at 100K tracked parameters.

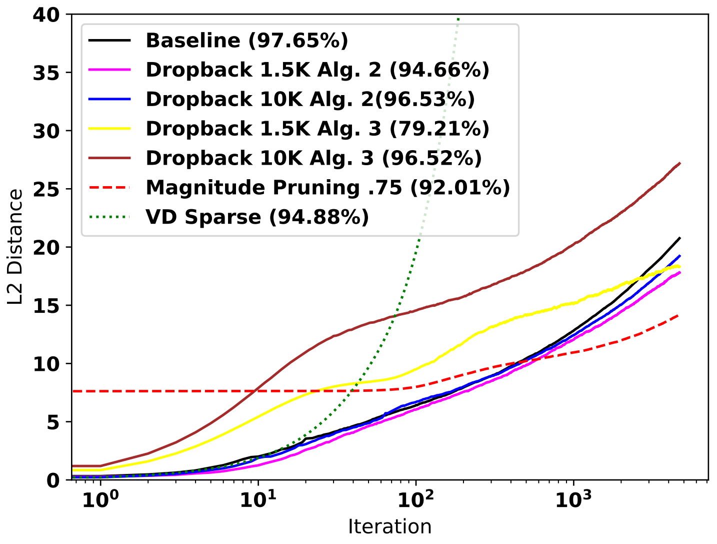


**Figure 4.19:** Diffusion $(\ell^2)$ distance vs. training time on MLP-100 (note log time scale).

## 4.5 Discussion

Finally, we sought to understand how DropBack is able to offer better accuracy at higher weight reduction ratios than prior work across a wide range of deep neural networks.

To investigate this, we applied the training process analysis from Hoffer et al. [23]. Briefly, the authors observe that the average $\ell^2$ (Euclidean) distance of weights from their initial values is a logarithmic function of training time, i.e., $\|\mathbf{w}_t - \mathbf{w}_0\| \sim \log t$. They therefore model SGD as a random walk on a random potential surface, which exhibits the same logarithmic distance effect (known as ultra-slow diffusion). The authors demonstrate that SGD configurations that preserve the ultra-slow diffusion effect result in models that generalize well.

We therefore asked whether DropBack follows the same principle. We reasoned that, because DropBack tracks the highest gradients, DropBack should preserve the largest contributors to the $\ell^2$ diffusion distance of the baseline training scheme. In addition, because most of the remaining gradients are close to zero (see Figure 3.1), we expected the $\ell^2$ diffusion distance to evolve similarly to that of the baseline

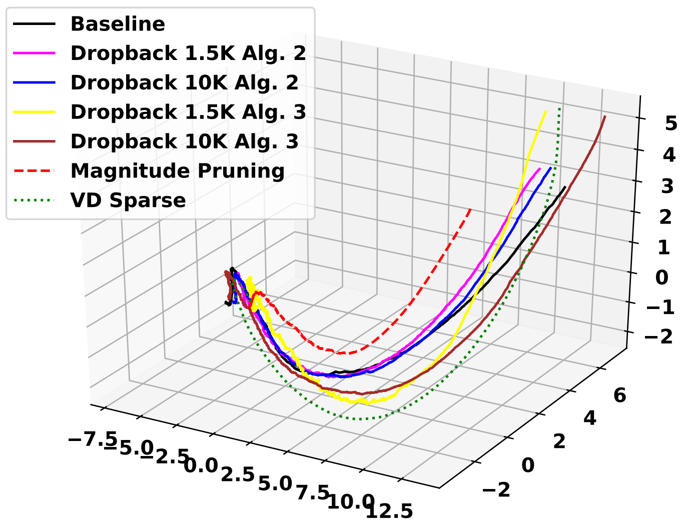


**Figure 4.20:** Evolution of weights under SGD projected into 3D space using PCA for DropBack, baseline, magnitude based pruning, and variational dropout.

unpruned training scheme. Finally, because fewer of the largest contributors are preserved when the weight reduction ratio is higher, we expected the average $\ell^2$ distance to be greater in configurations with a higher weight reduction ratio.

To verify this intuition, we measured the diffusion distance for the baseline uncompressed network, DropBack, variational dropout, and magnitude-based pruning, all on the MLP-100 network.[1] Figure 4.19 shows that under DropBack Algorithm 2 weights diffuse very similarly to the baseline training scheme. The overall $\ell^2$ distance is negligibly lower because the untracked weights remain at their initialization values. In contrast, magnitude-based pruning begins with a substantial $\ell^2$ distance (because many initialization values are immediately zeroed) and does not provide enough scaffolding structure for SGD to train well. Finally, variational dropout drastically alters the loss surface of the network, and so diffuses much faster than the baseline and DropBack. This results in numerical instability (the total variation of the solution is not bounded, and the optimization does not converge) as Hoffer et al. [23] predict;

[1]Network slimming, being a train-prune-retrain technique, is not amenable to this type of analysis.

this explains the failure of variational dropout to converge on the denser networks (see  Section 4.4).

To visualize how the weight values themselves evolve under DropBack compared to the Baseline and the two pruning techniques, we projected the parameter space to three dimensions using Principle Component Analysis (PCA). PCA is a dimensionality reduction technique that projects a coordinate space, in this case the parameters of the MLP-100 network, down to a space with fewer dimensions.

Figure 4.20 shows that under DropBack, the principal components of the trained weight vector stay very close to those of the Baseline-trained weight vector, whereas those of magnitude-based pruning and variational dropout diverge significantly. If we consider the training path of the Baseline configuration to be ideal, DropBack results in a similar near-ideal evolution.

With DropBack Algorithm 3, the diffusion distance is altered more significantly, and the path SGD takes during optimization changes correspondingly. When tracking only 1.5K of the original 90K parameters, the path and diffusion distance is altered drastically compared to the paths taken by Algorithm 2 and the Baseline. This explains the results from Table 4.2 and Table 4.4, where the DropBack variant Algorithm 2 performs better on the same dataset (MNIST) and network (MLP-100).

## 4.6 Summary

On the MNIST dataset, DropBack can reduce the number of weights in both LeNet-300-100 and MLP-100 to 20K tracked weights with validation errors of 1.78% and 1.70% respectively, compared to 1.41% error and 1.70% error for their respective unreduced baselines.

On the Fashion-MNIST dataset with VGG-S, DropBack can reduce the number of weights by 5× with a validation error of 5.43%, matching the baseline accuracy exactly.

On the CIFAR-10 dataset, DropBack is effective at reducing the number of weights used in all networks. On VGG-S, DropBack achieves a 9.90% validation error with a 5× weight reduction. On Densenet, DropBack achieves 5.86% validation error with 4.5× weight reduction, and on WRN-28-10 DropBack achieves 4.20% error with 7.3× weight reduction. For both Densenet and WRN-28-10 these weight reduction ratios are state of the art compared to post-training pruning techniques, without an increase in error.

Chapter 5

# Conclusion

Neural networks are becoming an everyday part of modern life. As their popularity grows, they are deployed to smaller and smaller devices. In order to fit the memory, energy, and computational bounds of these devices, neural networks must be compressed, which typically results in accuracy loss and longer training times. DropBack can reduce the number of weights in networks up to 5× with no accuracy loss and up to 36× with slight accuracy loss.

DropBack reduces weight storage both during and after training by (a) tracking only the weights with the highest accumulated gradients, and (b) recomputing the remaining weights on the fly. With the addition of a decay term for the untracked weight set, DropBack can be used to create compute sparsity both during and after training, increasing the potential for energy savings even more.

Because DropBack prunes precisely those weights that have learned the least, its weight diffusion profile during training is very close to that of standard (unconstrained) SGD, in contrast to other techniques. This allows DropBack to achieve better accuracy and weight reduction than prior methods on dense modern networks like Densenet and WRN-28-10, which have proven challenging to prune using existing techniques. On Densenet, a 5× weight reduction ratio can be achieved with no accuracy loss. On WRN-28-10, a 7× weight reduction ratio can be achieved with no accuracy loss. With a slight increase in error, WRN-28-10 can be reduced by 36×.

DropBack is the first pruning technique to store only a small subset of weight gradients during the training process. This is a key advantage, as it will make it possible for future work to dramatically reduce the memory footprint and memory bandwidth needed to store and access weights during training. Because of this, future designs that combines DropBack with custom hardware will be able to train networks larger than currently achievable with typical hardware, or to train standard-size networks on small mobile and embedded devices, which is currently not possible with standard training techniques and mobile GPUs.

# Bibliography

[1] A. F. Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018. URL http://arxiv.org/abs/1803.08375. → page 10

[2] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. URL http://arxiv.org/abs/1610.02132. → page 17

[3] J. M. Alvarez and M. Salzmann. Compression-aware Training of Deep Networks. *arXiv:1711.02638 [cs]*, Nov. 2017. arXiv: 1711.02638. → page 16

[4] M. Babaeizadeh, P. Smaragdis, and R. H. Campbell. NoiseOut: A simple way to prune neural networks. 2016. → pages 16, 31

[5] R. Brynjolfsson and A. Mcafee. The business of artificial intelligence. URL https://hbr.org/2017/07/the-business-of-artificial-intelligence. → page 1

[6] Z. Cai, X. He, J. Sun, and N. Vasconcelos. Deep Learning with Low Precision by Half-wave Gaussian Quantization. *arXiv:1702.00953 [cs]*, Feb. 2017. arXiv: 1702.00953. → page 17

[7] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv:1604.06174 [cs]*, Apr. 2016. arXiv: 1604.06174. → pages 4, 14

[8] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788, 2015. URL http://arxiv.org/abs/1504.04788. → pages 15, 17

[9] Y. Choi, M. El-Khamy, and J. Lee. Towards the Limit of Network Quantization. *arXiv:1612.01543 [cs]*, Dec. 2016. arXiv: 1612.01543. → page 16

[10] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. *arXiv:1412.7024 [cs]*, Dec. 2014. arXiv: 1412.7024. → page 17

[11] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]*, Feb. 2016. arXiv: 1602.02830. → page 17

[12] G. Cybenko. Approximation by superpositions of a sigmoidal function. math cont sig syst (mcss) 2:303-314. 2:303–314, 12 1989. → page 10

[13] T. Franck. Machine learning could lead to economic hypergrowth, new research suggests. URL https://www.cnbc.com/2017/10/21/machine-learning-could-lead-to-economic-hypergrowth-new-research-suggests.html. → page 1

[14] S. Ge, Z. Luo, S. Zhao, X. Jin, and X. Y. Zhang. Compressing deep neural networks for efficient visual inference. In *2017 IEEE International Conference on Multimedia and Expo (ICME)*, pages 667–672, July 2017. doi:10.1109/ICME.2017.8019465. → pages 1, 3, 16, 30, 34

[15] G. B. Goh, N. O. Hodas, and A. Vishnu. Deep learning for computational chemistry. URL http://arxiv.org/abs/1701.04503. → page 22

[16] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. 2015. → page 17

[17] P. Gysel. Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks. *arXiv:1605.06402 [cs]*, May 2016. arXiv: 1605.06402. → pages 16, 17

[18] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*, Oct. 2015. arXiv: 1510.00149. → pages 1, 3, 15, 16, 18, 22, 30

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016. → page 45

[20] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, B. Catanzaro, and W. J. Dally. DSD: Dense-Sparse-Dense Training for Deep Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2017. → page 38

[21] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks, 1993.*,, pages 293–299. IEEE, 1993. → pages 6, 15

[22] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. → page 11

[23] E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *NIPS*, 2017. → pages 50, 59, 60

[24] J. L. Holt and T. E. Baker. Back propagation simulations using limited precision calculations. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, pages 121–126 vol.2. → page 17

[25] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. doi:10.1109/ISSCC.2014.6757323. → pages 2, 3, 7, 25

[26] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely Connected Convolutional Networks. *arXiv:1608.06993 [cs]*, Aug. 2016. URL http://arxiv.org/abs/1608.06993. arXiv: 1608.06993. → page 35

[27] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. URL http://arxiv.org/abs/1608.06993. → pages xi, 33, 36

[28] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. 2016. → page 17

[29] D. A. Huffman. A method for the construction of minimum-redundancy codes. 40(9):1098–1101. ISSN 0096-8390. doi:10.1109/JRPROC.1952.273898. → page 18

[30] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50$\times$ fewer parameters and $<$0.5mb model size. *arXiv:1602.07360*, 2016. → page 34

[31] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*, 2014. → page 33

[32] D. P. Kingma, T. Salimans, and M. Welling. Variational dropout and the local reparameterization trick. 2015. → pages 16, 33

[33] A. Kingsley-Hughes. Inside Apple's new A11 Bionic processor. ZDNet, September 2017. → pages 3, 7

[34] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. → page 34

[35] J. Langford, L. Li, and T. Zhang. Sparse online learning via truncated gradient. 2008. → page 16

[36] Y. LeCun. The MNIST database of handwritten digits. 1998. → page 34

[37] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989. → pages 6, 10, 13, 21

[38] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990. → pages 3, 6, 15, 16, 20

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. 86(11):2278–2324, 1998. ISSN 0018-9219. → page 33

[40] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Effiicient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. → pages 12, 21, 24, 25, 26, 28

[41] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein. Training Quantized Nets: A Deeper Understanding. *arXiv:1706.02379 [cs, stat]*, June 2017. URL http://arxiv.org/abs/1706.02379. arXiv: 1706.02379. → pages 7, 34, 48

[42] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning Efficient Convolutional Networks through Network Slimming. *arXiv:1708.06519 [cs]*,

Aug. 2017. URL http://arxiv.org/abs/1708.06519. arXiv: 1708.06519. → pages 33, 46

[43] C. Louizos, M. Welling, and D. P. Kingma. Learning Sparse Neural Networks through l0 Regularization. *arXiv:1712.01312 [cs, stat]*, Dec. 2017. URL http://arxiv.org/abs/1712.01312. arXiv: 1712.01312. → pages 7, 34, 48

[44] J.-H. Luo, J. Wu, and W. Lin. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. *arXiv:1707.06342 [cs]*, July 2017. arXiv: 1707.06342. → pages 3, 16, 30, 34

[45] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. → page 18

[46] M. Masana, J. van de Weijer, L. Herranz, A. D. Bagdanov, and J. M. Alvarez. Domain-adaptive deep network compression. *arXiv:1709.01041 [cs]*, Sept. 2017. arXiv: 1709.01041. → pages 3, 14, 16, 30, 34

[47] D. Masters and C. Luschi. Revisiting small batch training for deep neural networks. URL http://arxiv.org/abs/1804.07612. → page 4

[48] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr. WRPN: Wide reduced-precision networks. 2017. → page 17

[49] D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. 2017. → pages 16, 31

[50] A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, and J. Martens. Adding gradient noise improves learning for very deep networks. URL http://arxiv.org/abs/1511.06807. → page 49

[51] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv:1603.05279 [cs]*, Mar. 2016. arXiv: 1603.05279. → page 17

[52] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *CoRR*, abs/1602.08124, 2016. URL http://arxiv.org/abs/1602.08124. → page 14

[53] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. *arXiv:1705.01626 [cs]*, May 2017. arXiv: 1705.01626. → page 14

[54] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL https://books.google.ca/books?id=P_XGPgAACAAJ. → page 10

[55] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi:10.1007/s11263-015-0816-y. → page 10

[56] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014. URL https://www.microsoft.com/en-us/research/publication/1-bit-stochastic-gradient-descent-and-application-to-data-parallel-distributed-training-of-speech-dnns/. → page 17

[57] P. Y. Simard and H. P. Graf. Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, pages 232–239, 1994. → page 17

[58] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556. → pages 33, 42

[59] A. Snyder. Behind the boom in machine learning. URL https://www.axios.com/behind-the-boom-in-machine-learning-1513388387-e5b2d881-f055-46d9-859c-c7d743b18c04.html. → page 1

[60] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. → pages 3, 16, 34

[61] C. V. M. L. Team. Personalized hey siri - apple, . URL https://machinelearning.apple.com/2018/04/16/personalized-hey-siri.html. → page 1

[62] C. V. M. L. Team. An on-device deep neural network for face detection - apple, . URL https://machinelearning.apple.com/2017/11/16/face-detection.html. → page 1

[63] S. Tokui, K. Oono, S. Hido, and J. Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in (NIPS)*, volume 5, 2015. → page 32

[64] K. Ullrich, E. Meeds, and M. Welling. Soft Weight-Sharing for Neural Network Compression. *arXiv:1702.04008 [cs, stat]*, Feb. 2017. arXiv: 1702.04008. → pages 1, 3, 16, 30

[65] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of neural networks using DropConnect. In *PMLR*, pages 1058–1066. → pages 3, 16

[66] J. Wangni, J. Wang, J. Liu, and T. Zhang. Gradient sparsification for communication-efficient distributed optimization. URL http://arxiv.org/abs/1710.09854. → page 17

[67] J. Wu, W. Huang, J. Huang, and T. Zhang. Error compensated quantized SGD and its applications to large-scale distributed optimization. URL http://arxiv.org/abs/1806.08054. → page 17

[68] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized Convolutional Neural Networks for Mobile Devices. *arXiv:1512.06473 [cs]*, Dec. 2015. arXiv: 1512.06473. → pages 1, 16

[69] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. 2017. → pages 34, 42

[70] S. Zagoruyko. Torch | 92.45% on CIFAR-10 in torch. URL http://torch.ch/blog/2015/07/30/cifar.html. → page 33

[71] S. Zagoruyko and N. Komodakis. Wide Residual Networks. *arXiv:1605.07146 [cs]*, May 2016. URL http://arxiv.org/abs/1605.07146. arXiv: 1605.07146. → pages xi, 2, 33, 35, 36

[72] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *arXiv:1702.03044 [cs]*, Feb. 2017. arXiv: 1702.03044. → pages 16, 17

[73] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]*, June 2016. arXiv: 1606.06160. → page 17

[74] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui. SparseNN: An Energy-Efficient Neural Network Accelerator Exploiting Input and Output Sparsity. *arXiv:1711.01263 [cs]*, Nov. 2017. arXiv: 1711.01263. → pages 3, 14, 16, 30

[75] M. Zhu and S. Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. 2017. → pages 16, 22, 31