Efficiency in Machine Learning with Focus on Deep Learning and Recommender Systems

by

Amy Nesky

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Professor Quentin Stout, Chair
Assistant Professor David Fouhey
Assistant Professor Danai Koutra
Associate Professor Barzan Mozafari
Professor Martin Strauss

Share, because boosted and ensemble methods are the best ones.

Amy Nesky

anesky@umich.edu

ORCID iD: 0000-0002-3454-7210

© Amy Nesky 2020

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256260. This work used the Extreme Science and Engineering Discovery Environment, which is supported by National Science Foundation grant number OCI-1053575. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF FIGURES v

LIST OF TABLES viii

LIST OF APPENDICES ix

ABSTRACT x

CHAPTER

I. Introduction 1

1.1 Background 1
1.2 Overview 2

II. Neural Networks with Block Diagonal Inner Product Layers 5

2.1 Related Work 7
2.2 Methodology 10
2.3 Experiments: Speedup and Accuracy 11
2.3.1 Speedup 12
2.3.2 Accuracy Results 16
2.4 Random matrix theory observations 22
2.4.1 MNIST 24
2.4.2 CIFAR10 30
2.5 Discussion 32

III. Predictor-Corrector Gradient Descent 36

3.1 Related Work 37
3.2 Methodology 38
3.3 Relationship to Nesterov's Accelerated Gradient 41

3.4 Experimental Results 44
3.4.1 SVHN 45
3.4.2 CIFAR10 46
3.5 Discussion 49

IV. Generating Artificial Core Users for Interpretable Condensed Data 51

4.1 Related Work 52
4.2 Methodology 54
4.3 Experimental Results 61
4.3.1 MovieLens 62
4.4 Discussion 68

V. Conclusion 69

APPENDICES 71

BIBLIOGRAPHY 90

LIST OF FIGURES

Figure

2.1 Speedup when performing matrix multiplication using an $n \times n$ weight matrix and batch size 100. (Left) Speedup when performing only one forward matrix product. (Right) Speedup when performing all three matrix products involved in the forward and backward pass in gradient descent. Both images in this figure share the same key. . . 13

2.2 Using batch size 100, upper bound on the ratio of the number of pruning iterations to the number of pure block iterations that will result in an overall training speedup when using IP-BDIP layers. . . 15

2.3 For each inner product layer in Lenet-5 (Left) and Cuda-convnet (Right): forward runtimes of block diagonal and CSR sparse formats, combined forward and backward runtimes of block diagonal format. Lenet-5 uses batch size 64, and Cuda-convnet uses batch size 100. . 15

2.4 (Left) The difference between the accuracy on the training set and the accuracy on an independent test set when training Krizhevsky's Cuda-convnet (Krizhevsky 2012a) on the CIFAR10 dataset. (Right) Accuracy over the difference between the accuracy on the training set and the accuracy on an independent test set when training Krizhevsky's Cuda-convnet (Krizhevsky 2012a) on the CIFAR10 dataset. 23

2.5 10000 training iterations using LeNet-5 net on MNIST. (Left) The ratio of the ip1 weight matrix variance in $(b_1, 1)$-BD$_4$ over the ip1 weight matrix variance in $(1, 1)$-BD$_4$. (Right) Ratio of trained ip1 weight matrix singular values over singular values of a truly random matrix with the same dimensions. 26

2.6 10000 training iterations using LeNet-5 net on MNIST. Expected value taken over order of the distance between 1 and the ratio of sorted, aggregated singular values of ip1 weight matrices for sister $(b_1, 1)$-BD$_4$ networks and the singular values of R, a random matrix with iid entries of equal dimension. 28

2.7 10000 training iterations using LeNet-5 net on MNIST. (Left) Variance of weight matrix entries in layer ip1 in both the fully connected and block diagonal setting. (Right) Singular values of ip1 weight matricies for sister $(b_1, 1)$-BD$_4$ networks and of R, a random matrix with iid entries of equal dimension. 28
2.8 $\lambda_{\frac{1}{500}W_{b_1}W_{b_1}^\top}$ is the measured empirical spectral distribution of $\frac{1}{500}W_{b_1}W_{b_1}^\top$ where W_{b_1} is the ip1 layer weight matrix in the $(b_1, 1)$-BD architecture after 10000 training iterations on MNIST using LeNet-5. Bar graph of $\lambda_{\frac{1}{500}W_1W_1^\top}$ with plot of $\lambda_{\frac{1}{500}RR^\top}$ for a random matrix R with the same variance. 30
2.9 9000 training iterations using Cuda-convnet on CIFAR10. (Left) Ratio of variance of ip1 weight matrix entries in block diagonal setting over variance of ip1 weight matrix entries in the fully connected setting. (Right) Ratio of trained ip1 weight matrix singular values over singular values of a truly random matrix with the same dimensions. 31
2.10 9000 training iterations using Cuda-convnet on CIFAR10. (Left) Variance of weight matrix entries in layer ip1 in both the fully connected and block diagonal setting. (Right) Singular values of ip1 weight matricies for sister $(b_1, 1)$-BD$_5$ networks and of a random matrix with iid entries of equal dimension. 33
2.11 $\lambda_{\frac{1}{64}W_{b_1}W_{b_1}^\top}$ is the measured empirical spectral distribution of $\frac{1}{64}W_{b_1}W_{b_1}^\top$ where W_{b_1} is the ip1 layer weight matrix in the $(b_1, 1)$-BD$_5$ architecture after 9000 training iterations on the CIFAR10 dataset using Cuda-convnet framework. Bar graph of $\lambda_{\frac{1}{64}W_1W_1^\top}$ with plot of $\lambda_{\frac{1}{64}RR^\top}$ for a random matrix R with the same variance. 33
3.1 (Left) Maximum accuracy results on the SVHN data set. Testing takes place every 50 training iterations (Right) Slope of Left Figure versus iterations. 47
3.2 Results on the CIFAR10 data set. (Left) Maximum Accuracy versus iterations. Testing takes place every 250 training iterations. (Right) Percent of SGD iterations each method took to reach a particular accuracy. 47
4.1 (Left) Singular Values of 13000 Core User Ratings Matrix where Core Users are selected from the ml-20m dataset (Harper and Konstan 2015) using the most competitive selection method: the method used in the second row of Table 4.1. Singular Values of an equally sparse matrix with random ratings as the non-zero values. There are 13000 singular values, where by convention lower order singular values have larger value. The x-axis is labeled as the order percent, so the ith singular value would have x-tick value $i/130$. (Right) Absolute Value of the difference between the curves in the Left Figure. 63

4.2 Item Vectors Testing Error of 13000 ACUs over iterations using the ml-20m dataset (Harper and Konstan 2015). For each test, I averaged the results of Algorithm 2 over 20 runs where each run was given 200 independent testing users and (Left) 2600/(Right) 650 ACU item vectors. In each run, I stopped Algorithm 2 when I reached a training error of 0.2. 65
4.3 Sparse Mean Error of 13000 ACUs using the ml-20m dataset (Harper and Konstan 2015). 67
4.4 650 largest singular values of 13000 trained ACU ratings matrix using the ml-20m dataset (Harper and Konstan 2015), compared to the largest singular values of the ratings matrix of 27000 Core Users (20% of the users in the complete dataset) and the largest singular values of the ratings matrix of 13000 Core Users (10% of the users in the complete dataset). 67

LIST OF TABLES

Table

2.1 Accuracy results on MNIST dataset with the LeNet-5 network, and the SVHN and CIFAR10 datasets with the Cuda-convnet network. . . 19

3.1 Results on the CIFAR10 data set. Percent of SGD iterations each method took to reach a particular accuracy. 48

4.1 Item Vectors Testing Error of 13000 Core Users collected with previously existing methods using the ml-20m dataset (Harper and Konstan 2015). I averaged the results of Algorithm 2 over 75 runs where each run was given 200 independent testing users and 2600 of the Core User Item Vectors, or $\approx 50\%$ of the singular value mass. In each run, I stopped Algorithm 2 when I reached a training error of 0.2. 63

LIST OF APPENDICES

Appendix

A. Lenet-5 72
B. Cuda-Convnet 77
C. Proof of Theorem 3.3.1 and Corollary 3.3.1.1 84
D. NAG-PCGD 88

ABSTRACT

Machine learning algorithms have opened up countless doors for scientists tackling problems that had previously been inaccessible, and the applications of these algorithms are far from exhausted. However, as the complexity of the learning problem grows, so does the computational and memory cost of the appropriate learning algorithm. As a result, the training process for computationally heavy algorithms can take weeks or even months to reach a good result, which can be prohibitively expensive. The general inefficiencies of machine learning algorithms is a significant bottleneck slowing the progress in application sciences. This thesis introduces three new methods of improving the efficiency of machine learning algorithms focusing on expensive algorithms such as neural networks and recommender systems. The first method discussed makes structured reductions of fully connected layers in neural networks, which causes speedup during training and decreases the amount of storage required. The second method presented is an accelerated gradient descent method called Predictor-Corrector Gradient Descent (PCGD) that combines predictor-corrector techniques with stochastic gradient descent. The final technique introduced generates Artificial Core Users (ACUs) from the Core Users of a recommendation dataset. Core Users condense the number of users in a recommendation dataset without significant loss of information; Artificial Core Users improve the recommendation accuracy of Core Users yet still mimic real user data.

CHAPTER I

Introduction

1.1 Background

To say that Machine Learning (ML) is a 'powerful tool' would be a gross over-simplification of this area's true potential. In this century we've seen problems many considered insurmountable gracefully divined by ML algorithms. The game of Go is considered one of the most complex board games; events in one part of the board can be influenced by seemingly unrelated conditions in a distant part of the board, and decisions made early in the game can determine the environment a hundred moves later. Prior to 2008, computer Go players were unable to defeat professional human players given the largest handicap possible (*AGA News: Kim Prevails Again In Man Vs Machine Rematch* Retrieved 2009-08-08; *Supercomputer with innovative software beats Go Professional* Retrieved 2008-12-19). When Google's DeepMind developed AlphaGo using deep learning and instructing AlphaGo to learn from playing itself for hundreds of millions of games, suddenly computer players were able to consistently beat professional human Go players without any handicap at all (Silver et al. 2016). In the 1980's Ernst Dickmanns and his team at the Bundeswehr University Munich used computer vision to develop the first self-driving car that didn't need specialized infrastructure (Delcker 2018). Since then, the reliability of autonomous cars has improved to a level at which they share the road with human drivers across the United

States encountering and reasoning through infinitely complex situations previously only reasoned through by people. Deep learning can be trained on medical scans to diagnose abnormalities ranging from cancers to eye diseases with as much accuracy as health care professionals and in some cases outperforming them by a small margin (Kononenko 2001; McDermott 2019). These examples only scratch the surface of the range in successful applications of machine learning algorithms. The answer to Alan Turing's question, 'Can machines think?' from 1950 (Turing 1950), is no longer a straightforward 'no', and it seems we are still far from reaching the application potential of many ML algorithms.

We are still far from achieving the apparently endless ML capabilities largely because the process of training machine learning algorithms remains somewhat mysterious to us and cannot be completely methodical. We have an ever growing collection of suggested techniques for varying circumstances, but generally training a machine learning algorithm involves a great deal of guess and check. When training a computationally heavy algorithm like a neural network, that can take days to run to completion (Krizhevsky, Sutskever, and G. E. Hinton 2012), this can mean the learning feedback loop can take months to reach a good result. Such a slow feedback loop makes these machine learning algorithms prohibitively expensive for most users.

Reducing the training time, and ultimately the time to complete the learning feedback loop, can make the most powerful learning algorithms accessible for more users across diverse applications. This dissertation presents three methods to improve the efficiency of ML training focusing on some of the most costly ML algorithms available: neural networks and recommender systems.

1.2 Overview

The second chapter of this dissertation will introduce a method to make structured reductions of neural network architecture while maintaining accuracy; reducing

the network architecture in a structured manner allows for training speedup unlike traditional iterative pruning which requires additional formatting overhead to run (Han, H. Mao, and Dally 2015). Specifically, I consider a modified version of the fully connected layers I call a block diagonal inner product (BDIP) layer. These modified layers have weight matrices that are block diagonal, turning a single fully connected layer into a set of densely connected neuron groups. The blocks can be achieved by either initializing a purely block diagonal weight matrix or by iteratively pruning off-diagonal block entries. This idea is a natural extension of group, or depthwise separable, convolutional layers. In this chapter I will also briefly discuss some interesting trends I saw in the weight distributions when comparing the original network to the resulting reduced network. I observe that, even after thousands of training iterations, inner product layers have singular value distributions that resemble that of truly random matrices with iid entries, and that each block in a BDIP layer behaves like a smaller copy. For network architectures differing only by the number of blocks in one inner product layer, the ratio of the variance of the weights remains approximately constant for thousands of iterations. That is, the relationship in structure is preserved in the parameter distribution.

The third chapter presents a general accelerated gradient descent method called Predictor-Corrector Gradient Descent (PCGD) that combines predictor-corrector techniques with stochastic gradient descent. PCGD can be used to train any learning model in which gradient descent is appropriate. By using a sparse history of model parameter values to make periodic predictions of future parameter values, PCGD aims to skip unnecessary training iterations. PCGD circumvents the need to store all historical parameter values relevant to a particular prediction by incrementally updating the prediction function; for this reason, PCGD is suitable for training models with many parameters such as neural networks. In my experiments using PCGD to train neural networks, PCGD cut the number of training epochs needed for a network

to reach a particular testing accuracy by nearly one half when compared to stochastic gradient descent (SGD). PCGD was also able to outperform, with some trade-offs, Nesterov's Accelerated Gradient (NAG).

Chapter IV moves to the world of recommender systems and data compression. Recent work has shown that in a dataset of user ratings on items there exists a group of Core Users who hold most of the information necessary for recommendation. This set of Core Users can be as small as 20 percent of the users. Core Users can be used to make predictions for out-of-sample users without much additional work. Since Core Users substantially shrink a ratings dataset without much loss of information, they can be used to improve recommendation efficiency. I propose a method, combining latent factor models, ensemble boosting and K-means clustering, to generate a small set of Artificial Core Users (ACUs) from real Core User data. My ACUs incur a small amount of additional memory storage when compared to real Core Users, but remain a reduction in memory storage compared to the original dataset. Artificial Core Users improve the recommendation accuracy of real Core Users while remaining good centroids for the complete recommendation dataset.

CHAPTER II

Neural Networks with Block Diagonal Inner Product Layers

Most modern successful networks are made up of many convolutional layers followed by one to a few dense, fully connected (FC) layers that learn non-linear functions of high-level features (K. He et al. 2015; Huang et al. 2016; Krizhevsky, Sutskever, and G. E. Hinton 2012; Simonyan and Zisserman 2014; Szegedy et al. 2015; Xie et al. 2016; Zeiler and Fergus 2013). The final fully connected layers are important to allow combinations and mixing of convolutional features, that had previously only observed local relationships. While fully connected layers fill this important role, they are generally relegated to the end of the network and used sparingly, because they are comparatively costly to store and run. Fully connected layers do not share any weights like their convolutional counterparts; for a fully connected layer with m nodes and n inputs, $O(mn)$ weights need storing and computing the layer output takes $O(mn)$ time. This expense has pushed using many fully connected layers out of favor, especially in situations where space is limited, such as on mobile devices. If the cost of fully connected layers could be improved, this could provide flexibility and open up new possibilities in the deep learning.

Ideally, efforts to reduce the memory requirements of fully connected layers would also lessen their computational demand, but often these competing interests force

a trade-off. My work addresses both memory and computational efficiency without compromise. Focusing my attention on the fully connected layers, I decrease their memory footprint, improve their runtime and begin to uncover the mechanism of inner product layers.

There are a variety of methods to condense large networks without much harm to their accuracy. One such technique that has gained popularity is pruning (Han, H. Mao, and Dally 2015; Han, Pool, et al. 2015; Reed 1993), but traditional pruning has disadvantages related to network runtime. Most existing pruning processes slow down network training, and the resulting condensed network is usually significantly slower to execute (Han, H. Mao, and Dally 2015). Sparse format operations require additional overhead that can greatly slow down performance unless one prunes nearly all weight entries, which can damage network accuracy.

Localized memory access patterns can be computed faster than non-localized lookups. By implementing block diagonal inner product (BDIP) layers in place of fully connected layers, I condense these layers in a structured manner that speeds up the final runtime and does little harm to the final accuracy. BDIP layers can be implemented by either initializing a purely block diagonal weight matrix or by initializing a fully connected layer and focusing pruning efforts off the diagonal blocks to coax the dense weight matrix into structured sparsity. The first method reduces the gradient computation time and hence the overall training time. The latter method retains higher accuracy and supports the robustness of networks to *shaping*. That is, pruning can be used as a mapping between architectures — in particular, a mapping to more convenient architectures. Depending on how many iterations the pruning process takes, this method may also speed up training.

I have converted a single fully connected layer into an ensemble of smaller inner product learners whose combined efforts form a stronger learner, in essence boosting the layer. These methods also bring artificial neural networks closer to the architec-

ture of biological mammalian brains, which have more local connectivity (Herculano-Houzel 2012).

Another link with my work and the mammalian brain is the relationship to random matrix theory. In neuroscience, synaptic connectivity is often represented by a matrix with entries drawn randomly from an appropriate distribution (Rajan 2010; Rajan and Abbott 2006). The distribution of the singular values of a large, random matrix behaves predictably according to the Marchenko-Pastur Law (Marchenko and Pastur 1967). I will show that this distribution also represents artificial neural activity matrices well after thousands of training iterations. This relationship allows us to compare the behavior of inner product layers in networks that have related structure, thereby uncovering a piece of the inner product layer "black box". Specifically, I observe that when varying the number of blocks in a layer, the initial ratio of the variance of the weights is preserved to first order after thousands of training iterations.

2.1 Related Work

There is an assortment of criteria by which one may choose which weights to prune. With any pruning method, the result is a sparse network that takes less storage space than its fully connected counterpart. Han et al. iteratively prune a network using the penalty method by adding a mask that disregards pruned parameters for each weight tensor (Han, Pool, et al. 2015). This means that the number of required floating point operations decreases, but the number performed stays the same. Furthermore, masking out updates takes additional time. Han et al. report the average time spent on a forward propagation after pruning is complete and the resulting sparse layers have been converted to CSR format; for batch sizes larger than one, the sparse computations are significantly slower than the dense calculations (Han, H. Mao, and Dally 2015).

More recently, there has been momentum in the direction of structured reduction

of network architecture. Node pruning preserves some structure, but drastic node pruning can harm the network accuracy and requires additional weight fine-tuning (T. He et al. 2014; Srinivas and Babu 2015). Veit, Wilber, and Belongie 2016 showed that in in networks trained with identity shortcut connections like ResNet (K. He et al. 2015) and its variants, one can drop some of the full layers of a trained network and still have comparable performance. However, this behavior does not transfer to networks trained without identity shortcut connections. Other approaches include storing a low rank approximation for a layer's weight matrix (T. N. Sainath et al. 2013) and training smaller models on outputs of larger models ('distillation") (G. Hinton, Vinyals, and Dean 2014). Group lasso expands the concept of node pruning to convolutional filters (Lebedev and Lempitsky 2016; Wen et al. 2016; Yuan and Lin 2006). That is, group lasso applies L_1-norm regularization to entire filters. Indeed Group lasso is exactly analogous to node pruning when reconceiving a fully connected layer as a convolutional layer, since convolutions have fully connected layers as a special case.

It has been shown that reducing the network precision does minimal harm to the network accuracy (Courbariaux, Yoshua Bengio, and David 2015; Gupta et al. 2015; Vanhoucke, Senior, and M. Z. Mao 2011), which improves both the memory and computation efficiency of a network without disturbing the network structure. Binarized networks on the other hand, which use binary values for activations and weights are generally less accurate than their more precise counterparts (Simons and Lee 2019). Altering the a network precision is orthogonal to most other structured network reductions and can be used in conjunction, including block diagonal inner product layers.

Structured efficient linear layers form linear layers as a composition of matrices (Ailon and Chazelle 2009; Cheng et al. 2015; L2, Sarlo, and Smola 2013; Moczulski et al. 2016; Sindhwani, T. Sainath, and Kumar 2015). Sidhawani et al. propose

structured parameter matrices characterized by low displacement rank that yield high compression rate as well as fast forward and gradient evaluation (Sindhwani, T. Sainath, and Kumar 2015). Their work focuses on toeplitz-related transforms of the FC layer weight matrix. However, speedup is generally only seen for compression of large weight matrices. In (Moczulski et al. 2016), Moczulski et al. form efficient linear layers, called ACDC layers, composed of diagonal matrices and the discrete cosine transform matrix.

Group, or depthwise separable, convolutions have been used in recent CNN architectures with great success (Chollet 2017; Ioannou et al. 2017; X. Zhang et al. 2017). In group convolutions, a particular filter does not see all of the channels of the previous layer. BDIP layers apply this idea of separable neuron groups to the FC layers. This method transforms a fully connected layer into an ensemble of smaller fully connected neuron groups that boost the layer. Again, it should be mentioned here that fully connected layers can be represented as 1×1 convolutions and so they can be thought of as a special case of convolutions. However, it is worth studying block diagonal inner product layers specifically because as a particularly costly special case that can be implemented using a regular matrix product, there is a lot of room for efficiency improvement both with regards to storage and computation. Additionally, fully connected layers jumble up local information unlike more general convolutional layers, and so it is important to understand the consequences of imposing additional structure.

There is less work considering the distribution of weights in artificial neural networks. Initialization distributions to combat vanishing gradients are supported by theoretical variances for back propagation gradients under the assumption that the weights are independent, which is not valid beyond the first iteration (Glorot and Y. Bengio 2010; K. H. X. Z. S. R. J. Sun 2015). Random weights have been looked at as good predictors of successful network architecture (Saxe et al. 2011). More recently,

N.Tishby discussed the trend of the distribution of weight updates as they relate to the Mutual Information Plane (N.Tishby 2017). To the best of my knowledge, the effects of architecture on the change in distribution of the weights through training and the connection between trained inner product layer weights and iid random matrices have not been explored. In theoretical neuroscience, random matrices are used to model synaptic connections and to study brain plasticity (Rajan 2010; Rajan and Abbott 2006; Sompolinsky, Crisanti, and Sommers 1988). Knowing that inner product layers in artificial neural networks are well modeled by random matrices opens the field to a new range of analytical tools that may support a specific network's robustness or plasticity.

2.2 Methodology

I consider two methods for implementing BDIP layers:

1. I initialize a layer with a purely block diagonal weight matrix and keep the number of connections constant throughout training.

2. I initialize a fully connected layer and iteratively prune entries off the diagonal blocks to achieve a block substructure.

When a BDIP layer is implemented using the second method we'll add the prefix 'IP', written IP-BDIP, to indicate it is an iteratively pruned BDIP layer. Within a layer, all blocks have the same size.[1] IP-BDIP layers are accomplished in three phases: a dense phase, an iterative pruning phase and a block diagonal phase. In the dense phase a fully connected layer is initialized and trained in the standard way. During the iterative pruning phase, focused pruning is applied to entries off the diagonal blocks using the weight decay method with L_1-norm. That is, if W is the weight

[1]In my work I chose to implement all blocks with the same size, but blocks do not need to have the same size in general.

matrix for a fully connected layer I wish to push toward block diagonal, I add

$$D = \alpha \sum_{i,j} |\mathbb{1}_{i,j} W_{i,j}| \tag{2.1}$$

to the loss function during the iterative pruning phase, where α is a tuning parameter, $W_{i,j}$ is an entry in the layer's weight matrix and $\mathbb{1}_{i,j}$ is an indicator function such $\mathbb{1}_{i,j} = 0$ when $W_{i,j}$ is off the diagonal blocks and $\mathbb{1}_{i,j} = 1$ when $W_{i,j}$ is in a diagonal block. When pruning is complete, to maximize speedup it is best to reformat the weight matrix once such that the blocks are condensed and adjacent in memory.[2] Batched smaller dense calculations for the blocks use cuBLAS strided batched multiplication (Nickolls et al. 1998). There is a lot of flexibility when using IP-BDIP layers that can be tuned for specific user needs. More pruning iterations may increase the total training time but can yield higher accuracy and reduce overfitting.

2.3 Experiments: Speedup and Accuracy

My goal is to reduce memory storage of the inner product layers while maintaining or reducing the final execution time of the network with minimal loss in accuracy. We will also see reduction of the total training time in some cases. All experiments are run on the Bridges' NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center. All computations are done with float precision.

For speedup analysis I timed block diagonal multiplications using $n \times n$ matrices with varying dimension sizes and varying numbers of blocks; I considered the forward pass and gradient updates. I also calculate an upper bound on the ratio of the number of pruning iterations to the number of pure block iterations that will yield speedup when using IP-BDIP layers.

For accuracy results, I ran experiments on three standard image classification

[2]When using BDIP layers, one should alter the output format of the previous layer and the expected input format of the following layer accordingly, in particular to row major ordering.

datasets: MNIST (LeCun, Cortes, and Burges n.d.), SVHN (Netzer et al. 2011) and CIFAR10 (Krizhevsky 2009). MNIST and SVHN are both digit classification datasets; MNIST is handwritten in black and white, and SVHN contains colored pictures taken from the street of house numbers. The CIFAR10 dataset contains low resolution, colored images of objects in ten classes. I ran experiments on the MNIST dataset using a LeNet-5 (LeCun, Bottou, et al. 1998) network, and the SVHN and CIFAR10 datasets using Krizhevsky's Cuda-convnet (Krizhevsky 2012a). Cuda-convnet does not produce state-of-art accuracies for SVHN or CIFAR10, but demonstrates the performance differences between my methods and others. I also ran a few experiments on smaller, purely inner product layer networks without convolutional or other types of layers. With interest favoring deeper convolutional nets, I dedicate more space in this paper to exploring BDIP layers in convolutional nets to demonstrate their compatibility with modern networks. I implement my work in Caffe, which provides these architectures; Caffe's MNIST example uses LeNet-5 and Cuda-convnet can be found in Caffe's CIFAR10 "quick" example. A detailed architecture outline for the LeNet-5 network can be found in Appendix A, and one for Krizhevsky's Cuda-convnet can be found in Appendix B.

For ease of transcription, let $(b_1, \ldots, b_n)$-BD$_m$ denote a network architecture with m layers, not including the input layer, in which the last n layers are BDIP layers, where $b_i = j$ indicates that the ith BDIP layer has j blocks along the diagonal. If $b_i = 1$ then the ith inner product layer is fully connected. In all cases in this paper, if $m > n$ then the first $m - n$ layers are convolutional.

2.3.1 Speedup

Figure 2.1 shows the speedup when performing matrix multiplication using an $n \times n$ weight matrix and batch size 100 when the weight matrix is purely block diagonal. In this section, speedup is always relative to the unaltered, fully connected

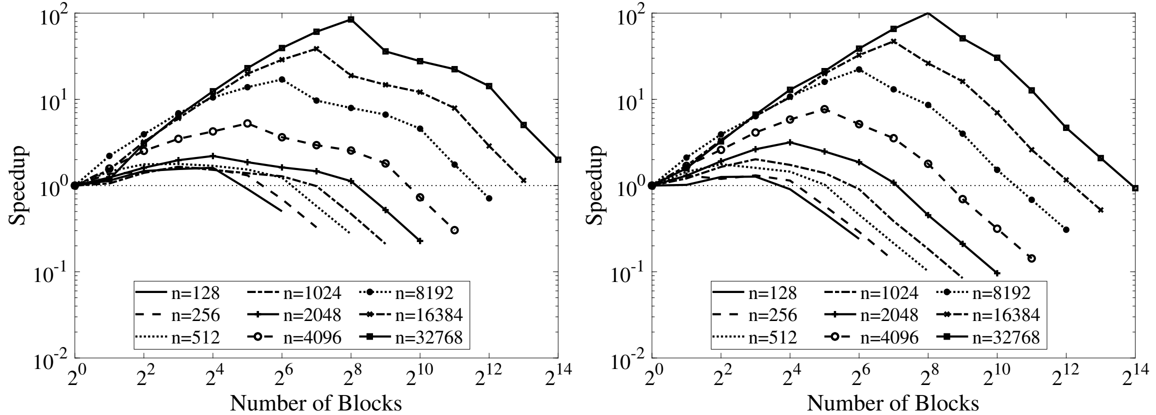

Figure 2.1: Speedup when performing matrix multiplication using an $n \times n$ weight matrix and batch size 100. (Left) Speedup when performing only one forward matrix product. (Right) Speedup when performing all three matrix products involved in the forward and backward pass in gradient descent. Both images in this figure share the same key.

calculation. The speedup when performing only the forward-pass matrix product is shown in the left pane, and the speedup when performing all gradient descent products is shown in the right pane. As the number of blocks increases, the overhead to perform cuBLAS strided batched multiplication can become noticeable; this library is not yet well optimized for performing many small matrix products (Masliah et al. 2016). However, with specialized batched multiplications for many small matrices, Jhurani et al. attain up to 6 fold speedup (Jhurani and Mullowney 2015). Using cuBLAS strided batched multiplication, maximum speedup is achieved when the number of blocks is 2^{-7} times the matrix dimension. When only timing the forward pass, the speedup is always greater than 1 when the number of blocks is at most 2^{-5} times the matrix dimension. When timing the forward and backward pass, the speedup is always greater than 1 when the number of blocks is at most 2^{-6} times the matrix dimension.

On the other hand, using toeplitz-related transforms, for displacement rank higher than approximately $2^{-9.5}$ times the matrix dimension the forward pass is slowed down, and backward pass is slowed down for displacement rank higher than approximately

$2^{-10.4}$ times the matrix dimension (Sindhwani, T. Sainath, and Kumar 2015). From Figure 3 in (Sindhwani, T. Sainath, and Kumar 2015), speedup is generally only seen for compression of large weight matrices. From Figure 2 in (Moczulski et al. 2016), We can see that ACDC layers do consistently provide speedup for multiple calls when compared to a dense linear layer. They achieve a maximum speedup of approximately 10 times for layers with dimension at most 8192, but in Figure 2.1, we can see that BDIP layers exceed this maximum speedup by a small but clear margin for layers with dimension at most 8192.

For a given inner product layer, using IP-BDIP layers we would see speedup in that layer's training time if

$$\frac{T(\mathrm{FC}) - T(\mathrm{Block})}{T(\mathrm{Prune})} > \frac{y}{x} \tag{2.2}$$

where $T(\cdot)$ is the combined time to perform the forward and backward passes of an inner product layer in the input state, x is the number of pure block iterations, and y is the number of pruning iterations. $T(\mathrm{Prune})$ is the time to regularize and apply a mask to the off diagonal block layer weights, which happens once in a training iteration. Figure 2.2 plots the upper bound in ratio 2.2 against the number of blocks for a layer with an $n \times n$ weight matrix and batch size 100.

Figure 2.3 shows timing results for the inner product layers in LeNet-5 (Left) and Cuda-convnet (Right), which both have two inner product layers. I plot the forward runtime speedup per inner product layer when the layers are purely block diagonal, the combined forward and backward runtime speedup to do the three matrix products involved in gradient descent training when the layers are purely block diagonal, and the runtime speedup of sparse matrix multiplication with random entries in CSR format using cuSPARSE (Nickolls et al. 1998). The points at which the forward sparse and forward block curves meet in each plot in Figure 2.3 indicate the FC

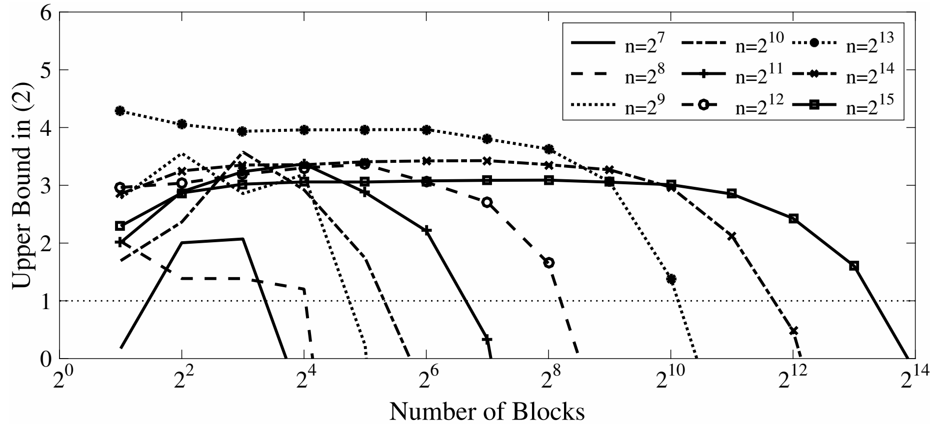

Figure 2.2: Using batch size 100, upper bound on the ratio of the number of pruning iterations to the number of pure block iterations that will result in an overall training speedup when using IP-BDIP layers.

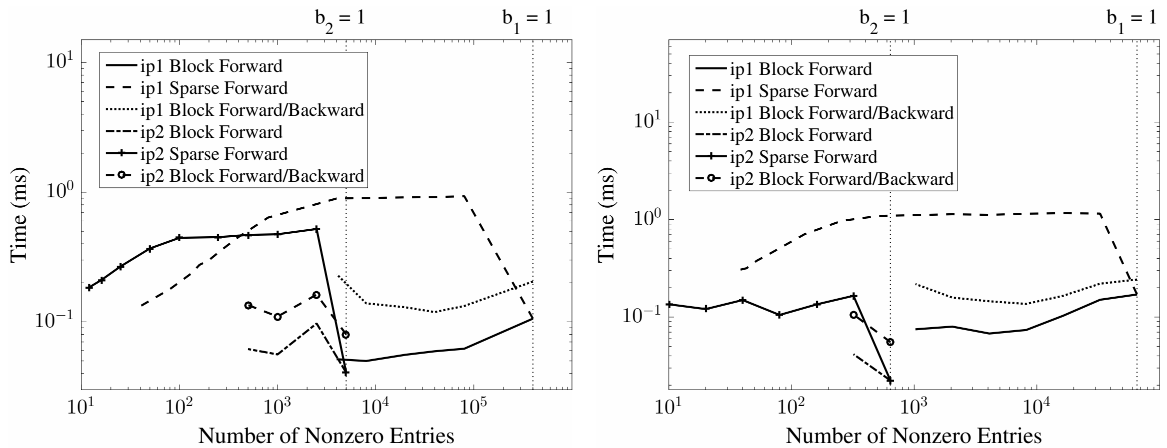

Figure 2.3: For each inner product layer in Lenet-5 (Left) and Cuda-convnet (Right): forward runtimes of block diagonal and CSR sparse formats, combined forward and backward runtimes of block diagonal format. Lenet-5 uses batch size 64, and Cuda-convnet uses batch size 100.

dense forward runtime speedups for each layer; these are made clearer with dotted, black, vertical lines. Note that the block forward and combined forward/backward curves almost perfectly overlap in Figure 2.3 (Right).

In LeNet-5, the first inner product layer, ip1, has a 500×800 weight matrix, and the second inner product layer, ip2, has a 10×500 weight matrix, so the (b_1, b_2)-BD$_4$ LeNet-5 architecture has $(800 \times 500)/b_1 + (500 \times 10)/b_2$ nonzero weights across both inner product layers. Figure 2.3 (Left) shows there is greater than 1.4 times speedup for greater than or equal to 8000 nonzero entries in ip1, which happens for $b_1 \leq 50$, when timing both forward and backward matrix products in (b_1, b_2)-BD$_4$ LeNet-5, and 1.6 times speedup when $b_1 = 100$, or 4000 nonzero entries, when only timing the forward matrix product in (b_1, b_2)-BD$_4$ LeNet-5.

In Cuda-convnet, the first inner product layer, ip1, has a 64×1024 weight matrix, and the second inner product layer, ip2, has a 10×64 weight matrix. The (b_1, b_2)-BD$_5$ Cuda-convnet architecture has $(1024 \times 64)/b_1 + (64 \times 10)/b_2$ nonzero entries across both inner product layers. Figure 2.3 (Right) shows there is greater than 1.26 times speedup for greater than or equal to 2048 nonzero entries in ip1, which happens for $b_1 \leq 32$, when timing both forward and backward matrix products in (b_1, b_2)-BD$_5$ Cuda-convnet, and 1.65 times speedup for greater than or equal to 1024 nonzero entries in ip1, which happens for $b_1 \leq 64$, when only timing the forward matrix product in (b_1, b_2)-BD$_5$ Cuda-convnet.

In both plots of Figure 2.3 we see sparse format performs poorly. Sparse format can be more than 8 times slower than dense calculations.

2.3.2 Accuracy Results

All hyperparameters and initialization distributions provided by Caffe's example architectures are left unchanged (see Appendices A and B). Training is done with batched gradient descent using the cross-entropy loss function on the softmax of

the output layer. In my experiments I performed only manual tuning of the new hyperparameter introduced by IP-BDIP layers (see equation 2.1).

In ShuffleNet, Zhang et al. note that when multiple group convolutions are stacked together this can block information flow between channel groups and weaken representation (X. Zhang et al. 2017). To correct for this, they suggest dividing the channels in each group into subgroups, and shuffling the outputs of the subgroups in this layer before feeding them to the next layer. Applying this approach to block inner product layers requires either moving entries in memory or doing more, smaller matrix products. Both of these options would hurt efficiency.

Using IP-BDIP layers also addresses information flow. Pruning does add some work to the training iterations, but, unlike the ShuffleNet method, does not add work to the final execution of the trained network. After pruning is complete, the learned weights are the result of a more complete picture; while the information flow has been constrained, it is preserved as an artifact in the remaining weights. Another alternative is to randomly shuffle whole blocks each pass like in the "random sparse convolution" layer in the CNN library *cuda-convnet* (Krizhevsky 2012b), not to be confused with the network architecture by the same name. I found that for the inner product layers in LeNet-5 and Krizhevsky's Cuda-convnet network, the ShuffleNet method did not show as much improvement in accuracy as randomly shuffling the whole blocks, so I do not include results using the ShuffleNet method.

Table 2.1 shows the accuracy results for BDIP layers, BDIP layers with random block shuffling, IP-BDIP layers and layers with traditional iterative pruning using the penalty method to prune weight entries not subject to any confinement or organization. The baseline accuracy without using any parameter-efficient layers can be found in parenthesis next to the dataset name in Table 2.1.[3] I show accuracy results for the most condensed net with BDIP layers and the net with the fastest speedup in

[3]Here I denote the baseline architecture using the notation $(1, \ldots, 1)$-BD$_m$.

the inner product layers.

2.3.2.1 MNIST

I experimented on the MNIST dataset with the LeNet-5 framework (LeCun, Bottou, et al. 1998) using a training batch size of 64 for 10000 iterations. LeNet-5 has two convolutional layers with pooling followed by two inner product layers with ReLU activation (see Appendix A for an architecture outline and hyperparameter details). LeNet-5 (1,1)-BD_4 achieves a final accuracy of 99.11%. In all cases testing accuracy remains within 1% of this (1,1)-BD_4 accuracy.

When training BDIP layers using implementation method 2 (with pruning), I perform 15 dense calculations with L_1-regularization off the diagonal blocks using regularization coefficient $\alpha = 0.005$. After 15 dense iterations, this implementation method then forces entries with modulus less than 0.05 to zero every fifth iteration until at most 0.1% of the off-diagonal block entries survive, at which point no more structured regularization is done. The inner product layers weights are initialized using the Xavier weight filler Glorot and Y. Bengio 2010, which samples a uniform distribution with variance $1/n_{in}$, where n_{in} is the number of neurons feeding into a node. Using implementation method 2 (with pruning), for (b_1, b_2)-BD_4 with $b_1 \leq 50$, the ip1 layer weights are initialized with variance $b_1/800$; this initialization variance mirrors the pure block diagonal initialization even though the matrix begins as fully connected. Using implementation method 2 (with pruning), for $(100, b_2)$-BD_4 the ip1 layer weights are initialized with variance 60/800 to prevent instability.For all implementation methods, I initialized weights in ip2 with variance 1/500.

Using traditional iterative pruning with L_2 regularization, as suggested in (Han, Pool, et al. 2015), pruning every fifth iteration until 4000 and 500 nonzero entries survived in ip1 and ip2 respectively gave an accuracy of 98.55%, but the resulting forward multiplication runtime was more than 8 times slower than the dense FC case

Table 2.1: Accuracy results on MNIST dataset with the LeNet-5 network, and the SVHN and CIFAR10 datasets with the Cuda-convnet network.

	BDIP	**rand. shuff**	**IP-BDIP**	**trad. prune**
MNIST (99.11% **Accurate when using (1,1)-BD$_4$**)				
(10,1)-BD$_4$	98.83%	98.81%	99.02%	99.04%
(100,10)-BD$_4$	98.39%	98.42%	98.65%	98.55%
SVHN (91.96% **Accurate when using (1,1)-BD$_5$**)				
(8,1)-BD$_5$	91.39%	91.46%	91.88%	91.15%
(64,2)-BD$_5$	89.21%	89.69%	90.02%	90.93%
CIFAR10 (76.29% **Accurate when using (1,1)-BD$_5$**)				
(8,1)-BD$_5$	75.07%	75.09%	76.05%	75.64%
(64,2)-BD$_5$	72.7%	73.45%	74.81%	75.18%

(See Figure 2.3 Left). On the other hand, implementing the LeNet-5 $(100, 10)$-BD$_4$ architecture with IP-BDIP layers using 15 dense iterations and 350 pruning iterations gave a final accuracy of 98.65%. In this case, the ratio of the number of pruning iterations to the number of pure block iterations is ≈ 0.04. In this setting, using neither random shuffling of whole blocks in ip1, nor fixed sub-block shuffling in the manner of X. Zhang et al. 2017 delivered any noticeable improvement. (10,1)-BD$_4$ yielded approximately 1.4 times speedup for all gradient descent matrix products in both inner product layers after any pruning is complete, and (100,10)-BD$_4$ condensed the inner product layers in LeNet-5 approximately 81 fold.

In (Sindhwani, T. Sainath, and Kumar 2015), Toeplitz (3) has error rate 2.09% using a single hidden layer net with 1000 hidden nodes on MNIST. This method yields 63.32 fold compression over the FC setting. However from their Figure 3, this slows down the forward pass by around 1.5 times and the backward pass by around 5.5 times. A $(49, 1)$-BD$_2$ net with one hidden layer that has 980 hidden nodes has 29.43 fold compression and error rate 4.37% using IP-BDIP layers on MNIST. The speedup with this net is 1.53 for forward only and 1.04 when combining the forward and backward runtime. My net achieves less than a 5% error rate even though the

blocks of neurons in the hidden layer can only see a portion of the test input images.[4] A $(4, 1)$-BD$_2$ net with one hidden layer that has 1000 hidden nodes has 3.84 fold compression and error rate 2.12% using IP-BDIP layers on MNIST. What Toeplitz (3) gains in compression and accuracy, it sacrifices in execution time.

2.3.2.2 SVHN

I experimented on the SVHN dataset with Krizhevsky's Cuda-convnet (Krizhevsky 2012a) using batch size 100 for 9000 iterations. Krizhevsky's Cuda-convnet has three convolutional layers with ReLu activation and pooling, followed by two FC layers with no activation (see Appendix B for an architecture outline and hyperparameter details). Cuda-convnet (8,1)-BD$_5$ yielded approximately 1.5 times speedup for all gradient descent matrix products in both inner product layers when purely block diagonal, and Cuda-convnet (64,2)-BD$_5$ condensed the inner product layers in Cuda-convnet approximately 47 fold.

Using Cuda-convnet (1,1)-BD$_5$ I obtained a final accuracy of 91.96%. Table 2.1 shows all methods stayed under a 2.5% drop in accuracy. Using traditional iterative pruning with L_2 regularization pruning every fifth iteration until 1024 and 320 nonzero entries survived in the final two inner product layers respectively gave an accuracy of 90.93%, but the forward multiplication was more than 8 times slower than the dense FC computation. On the other hand, implementing Cuda-convnet $(64, 2)$-BD$_5$ with IP-BDIP layers, which has corresponding numbers of nonzero entries, with 500 dense iterations and less than 1000 pruning iterations gave a final accuracy of 90.02%. This is approximately 47 fold compression of the inner product layer parameters with only a 2% drop in accuracy when compared to (1,1)-BD$_5$.

[4]With more complex datasets, BDIP layers, especially without any pruning or block shuffling to assist information flow, are more appropriate in deeper layers.

2.3.2.3 CIFAR10

I experimented on the CIFAR10 dataset with Krizhevsky's Cuda-convnet (Krizhevsky 2012a) using batch size 100 for 9000 iterations. Krizhevsky's Cuda-convnet has three convolutional layers with ReLu activation and pooling, followed by two FC layers with no activation (see Appendix B for an architecture outline and hyperparameter details). I perform 15 dense calculations with L_1-regularization off the diagonal blocks using regularization coefficient $\alpha = 0.0075$. Let n_i be the number of off-diagonal block entries at initialization in inner product layer i. This method forces entries in inner product layer i with modulus less than 0.075 to zero every fifth iteration until at most $\min\{(n_i/2000)^2, 1000\}$ of the off-diagonal block entries survive, allowing more entries to survive as n_i grows. For example, for $b_1 = 64$, $n_1 = 64,512$ and after training the number of entries off the diagonal blocks is reduced by 98%. In all methods the inner product layer weights are initialized using a Gaussian filler with standard deviation 0.1, as suggested by Caffe. Using Cuda-convnet (1,1)-BD$_5$ I obtained a final accuracy of 76.29% after 9000 training iterations, which aligns with Caffe's reported accuracy using fully connected inner product layers. Table 2.1 shows all methods stayed within a 4% drop in accuracy. Using traditional iterative pruning with L_2 regularization pruning every fifth iteration until 1024 and 320 nonzero entries survived in the final two inner product layers gave an accuracy of 75.18%, but again the forward multiplication was more than 8 times slower than the dense FC computation. On the other hand, implementing Cuda-convnet $(64, 2)$-BD$_5$ with IP-BDIP layers, which has corresponding numbers of nonzero entries, with 500 dense iterations and less than 1000 pruning iterations gave a final accuracy of 74.81%. This is approximately 47 fold compression of the inner product layer parameters with only a 1.5% drop in accuracy. The total forward runtime of ip1 and ip2 in Cuda-convnet $(64, 2)$-BD$_5$ is 1.6 times faster than in (1,1)-BD$_5$. To achieve comparable speed with sparse format I used traditional iterative pruning to leave 37 and 40 nonzero entries

in the final inner product layers giving an accuracy of 73.01%. Thus implementing BDIP layers with pruning yields comparable accuracy and memory condensation to traditional iterative pruning with faster final execution time.

Whole node pruning decreases the accuracy more than corresponding reductions in the block diagonal setting. Node pruning until ip1 had only 2 outputs, i.e. a 1024×2 weight matrix, and ip2 had a 2×10 weight matrix for a total of 2068 weights between the two layers gave a final accuracy of 59.67%. On the other hand, using IP-BDIP layers, Cuda-convnet (64,2)-BD_5 has a total of 1344 weights between the two inner product layers and had a final accuracy 74.81%.

The final accuracy on an independent test set was 76.29% on CIFAR10 using the Cuda-convnet (1,1)-BD_5 net while the final accuracy on the training set itself was 83.32%. Using the Cuda-convnet (64,2)-BD_5 net without pruning, the accuracy on an independent test set was 72.49%, but on the training set was 75.63%. Figure 2.4 graphs the difference between the accuracy on the training set and the accuracy on an independent test set when training Krizhevsky's Cuda-convnet (Krizhevsky 2012a) on the CIFAR10 dataset using BDIP layers; I plot $(b_1, 1)$-BD_5 for various values of b_1. Figure 2.4 plots the ratio of the accuracy over the curves in Figure 2.4; we can see that more blocks yield a higher accuracy to overfit ratio. With IP-BDIP layers, the accuracy of (64,2)-BD_5 on an independent test set was 74.81%, but on the training set was 76.85%. Both block diagonal methods decrease overfitting, but IP-BDIP layers decreases overfitting slightly more.

2.4 Random matrix theory observations

The Marchenko-Pastur distribution describes the asymptotic behavior of the singular values of large random matrices with iid entries (Marchenko and Pastur 1967). Let X be an $m \times n$ matrix with iid entries x_{ij} such that $\mathrm{E}[x_{ij}] = 0$ and $\mathrm{Var}[x_{ij}] = \sigma^2$. The Marchenko-Pastur theorem states that as $n, m \to \infty$ such that $m/n \to y > 0$,

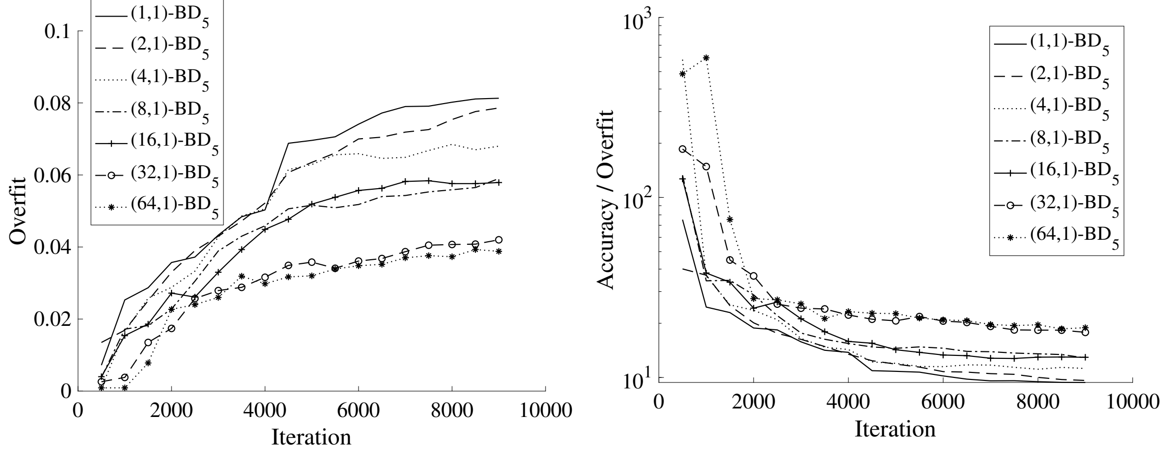

Figure 2.4: (Left) The difference between the accuracy on the training set and the accuracy on an independent test set when training Krizhevsky's Cuda-convnet (Krizhevsky 2012a) on the CIFAR10 dataset. (Right) Accuracy over the difference between the accuracy on the training set and the accuracy on an independent test set when training Krizhevsky's Cuda-convnet (Krizhevsky 2012a) on the CIFAR10 dataset.

with probability 1 the empirical spectral distribution of $\frac{1}{n}XX^\top$ converges in distribution to the density

$$\mu_y(x) = \begin{cases} \frac{1}{2\pi\sigma^2 yx}\sqrt{(b-x)(x-a)} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

with point mass $1-1/y$ at the origin if $y > 1$ where $a = \sigma^2(1-y^2)$ and $b = \sigma^2(1+y^2)$.

Network weights do not remain independent through training. Without momentum, weight parameter W receives the update $W \leftarrow W - \frac{\lambda}{b}\sum_{i=1}^{b}\frac{\partial L(x_i)}{\partial W}$ in an iteration, where λ is the learning rate, b is the batch size, L is the loss function and x_i is sampled from the data distribution. One can easily verify,

$$\begin{aligned} \Delta\text{Var}(W) =& \lambda^2\text{Var}\left(\frac{1}{b}\sum_{i=1}^{b}\frac{\partial L(x_i)}{\partial W}\right) \\ & - 2\lambda\text{Cov}\left(W, \frac{1}{b}\sum_{i=1}^{b}\frac{\partial L(x_i)}{\partial W}\right) \end{aligned} \tag{2.4}$$

using estimators for the right side of the equation. In my experience, the covariance quickly became the dominating term. However, for a large enough weight matrix, the singular values of an inner product layer weight matrix behave according to the Marchenko-Pastur distribution even after thousands of training iterations. That is, after thousands of correlated updates, the weight matrix behaves like a matrix with random iid entries. The assumption of independence fails at the micro level when calculating the change in variance of the weights, but is accurate at the macro level when considering the behavior of the weight matrix as an operator.

In this section I discuss only the first method for implementing BDIP layers without pruning. Networks that differ only by the number of blocks in one layer are referred to as sister networks. While a relationship between FC layer weights and the corresponding BDIP layer weights in trained sister networks is not evident from equation (2.4), indeed a relationship can be seen in the singular values of the weight matrices, and, in particular, in the change in the variance of the weight matrix entries throughout training. The initialization ratio of variances in corresponding layer weight matrices of sister networks persists through thousands of training iterations. This finding may provide a good mechanism for examining related network architectures and may support claims about a network's malleability or fitness.

2.4.1 MNIST

In my experiments on the MNIST dataset with the LeNet-5 framework (LeCun, Bottou, et al. 1998), the inner product layer weights are initialized using the Xavier algorithm (Glorot and Y. Bengio 2010). Thus the initialization variance of the weights in ip1 is $b_1/800$ if ip1 is a BDIP layer, where b_1 is the number of blocks, and the ratio of the ip1 initialization variance in $(b_1, 1)$-BD_4 LeNet-5 over the ip1 initialization variance $(1, 1)$-BD_4 LeNet-5 is just b_1. Figure 2.5 (Left) shows that this ratio is a good first order estimate for the final ip1 variance ratio after 10000 iterations in sister $(b_1, 1)$-

BD_4 LeNet-5 networks. I have written $(b_1, 1)$-$BD_4/(1, 1)$-BD_4 in the figure legend, but by this I mean the ratio of the ip1 weight matrix variance in $(b_1, 1)$-BD_4 LeNet-5 over that of $(1, 1)$-BD_4 LeNet-5. The final ip1 variance ratios are $5.03, 9.97, 19.96, 49.32$ and 96.99 for $b_1 = 5, 10, 20, 50$, and 100 respectively. We can see that the relationship deteriorates as the number of blocks increases. This phenomenon persists when the sigmoid activation function is used for layer ip1, keeping the activation function consistent across sister networks. When using the sigmoid activation function, the ratio seemed to deteriorate less quickly; e.g. the final ip1 variance ratio was 101.00 for $b_1 = 100$.

Figure 2.5 (Right) compares the singular values of the ip1 layer weight matrix in sister $(b_1, 1)$-BD_4 networks after 10000 training iterations to the singular values of a truly random 800×500 matrix who's entries were initialized with variance 6.6×10^{-4}, the final variance of the $(1, 1)$-BD_4 LeNet-5 ip1 layer weights. I denote the random matrix R. To make this comparison, I aggregate the singular values of each block along the diagonal in ip1 of $(b_1, 1)$-BD_4 LeNet-5 and sort them, but I note that the individual block spectral distributions appear identical to each other. For an array of singular values arranged by order, division is done entry-wise. I have written $(b_1, 1)$-BD_4/R in the figure legend, but by this I mean entry-wise division of the ordered singular values of the ip1 weight matrix in $(b_1, 1)$-BD_4 LeNet-5 over the ordered singular values of R. By convention, the lowest order singular values are the largest.

The individual curves in Figure 2.5 (Right) are difficult to distinguish. In fact, for each curve in Figure 2.5 (Right) the average distance from one, when averaging over order, is bounded above by 4×10^{-2} (see Figure 2.6). This behavior aligns with what the Marchenko-Pastur theorem dictates would happen to the ratio of the spectral distributions if the ip1 layer weight matrix had random iid entries, but in fact the ip1 layer weight matrices hold knowledge and are the product of 10000 correlated updates. By the Marchenko-Pastur theorem, increasing the variance of the entries in

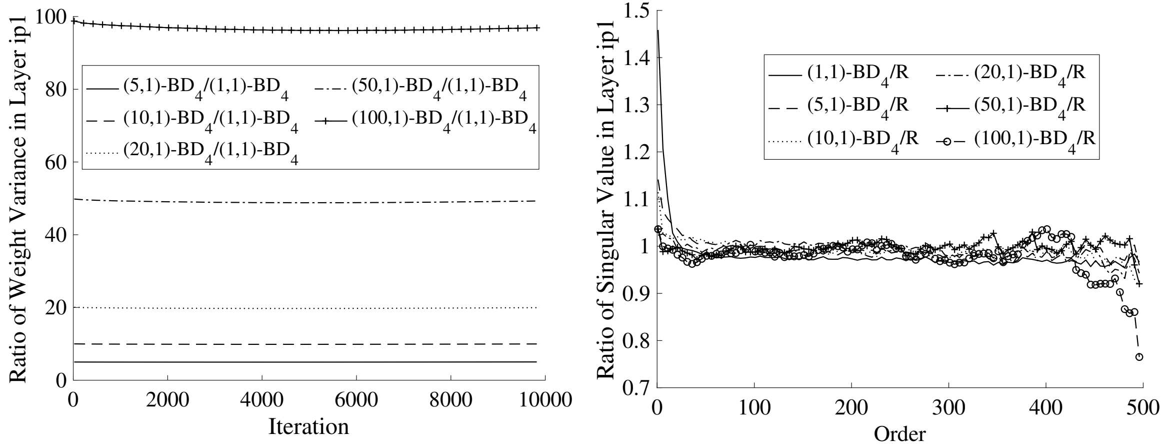

Figure 2.5: 10000 training iterations using LeNet-5 net on MNIST. (Left) The ratio of the ip1 weight matrix variance in $(b_1, 1)$-BD$_4$ over the ip1 weight matrix variance in $(1, 1)$-BD$_4$. (Right) Ratio of trained ip1 weight matrix singular values over singular values of a truly random matrix with the same dimensions.

a random matrix and simultaneously decreasing the matrix dimension by the same factor will not affect the singular value distribution; the decrease in matrix size would cancel with the increase in variance by the same factor (see equation (2.3)). Figure 2.5 (Left) shows that the ratio of the ip1 initialization variance in $(b_1, 1)$-BD$_4$ LeNet-5 over the ip1 initialization variance $(1, 1)$-BD$_4$ LeNet-5 is still b_1 after 10000 training iterations, and for b_1 blocks the ip1 layer weight matrix decreases in dimension by a factor of b_1. Figure 2.5 (Right) shows that these factors canceled for the trained ip1 matrices like they would for random matrices since the ratio of the singular values is just one. That is, the learned ip1 layer weight matrices behave like random operators.

The singular values of the trained ip1 layer weight matrices from $(b_1, 1)$-BD$_4$ LeNet-5 follow the curve that the singular values of the truly random matrix create with some error in the largest and smallest singular values. The disparity in the extreme singular values will be the focus of future work; it may be the first place where network learning become evident, or it may be the result of overfitting. Using $(1, 1)$-BD$_4$ LeNet-5, the accuracy reaches 98.28% by iteration 1000. After 1000 iterations, the ratio of the largest singular value of the trained weight matrix in layer ip1

over the largest singular value of a truly random matrix with the same dimensions and variance is 1.16. Figure 2.5 (Right) shows that this ratio is 1.44 after the full 10000 iterations. Figure 2.6 shows the expected value, taken over singular value order, of the distance between 1 and the ratio of sorted, aggregated singular values of ip1 weight matrices for sister $(b_1, 1)$-BD$_4$ LeNet-5 networks and the singular values of R, a random matrix with iid entries of equal dimension; this is plotted over training iterations for varying values of b_1. In Figure 2.6, we can see that at iteration zero smaller b_1 values correspond to expected ratios closer to one, which can be explained by the necessity of the limit in the Marchenko-Pastur theorem. On the other hand, after 10000 training iterations, smaller b_1 values correspond to expected ratios farther away from one indicating that larger values of b_1 maintain a final distribution that is more similar to that of a random matrix. In Figure 2.4 and 2.4, we learned that larger values of b_1 also correspond to reduced overfit.

The ratios in Figure 2.5 best highlight the relationship to random matrix theory and the relationship between ip1 layer weight matrix distributions in sister $(b_1, 1)$-BD$_4$ networks, but I also include the ip1 layer weight matrix variances and their singular values without taking a ratio in Figure 2.7. Figure 2.7 (Left) shows that the variances did change through training and so the fact that they maintained their original variance ratios is nontrivial. The curves in Figure 2.7 (Right) are again difficult to distinguish, but one can more clearly see the classic Marchenko-Pastur distribution.

Figure 2.8 (Left) compares the probability density function of the singular values of R, a truly random matrix with independent entries, to the measured distribution of the ip1 layer weight matrix singular values for the $(1, 1)$-BD$_4$ LeNet-5 architecture after 10000 training iterations. For a matrix M, λ_M is the PDF of the eigenvalues of M. I use W_1 to denote the ip1 layer weight matrix in $(1, 1)$-BD$_4$ after 10000 training iterations, and again R to denote a 800×500 random matrix with iid entries that

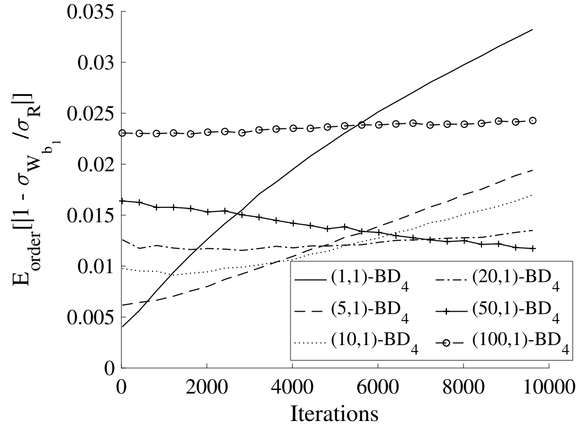

Figure 2.6: 10000 training iterations using LeNet-5 net on MNIST. Expected value taken over order of the distance between 1 and the ratio of sorted, aggregated singular values of ip1 weight matrices for sister $(b_1, 1)$-BD$_4$ networks and the singular values of R, a random matrix with iid entries of equal dimension.

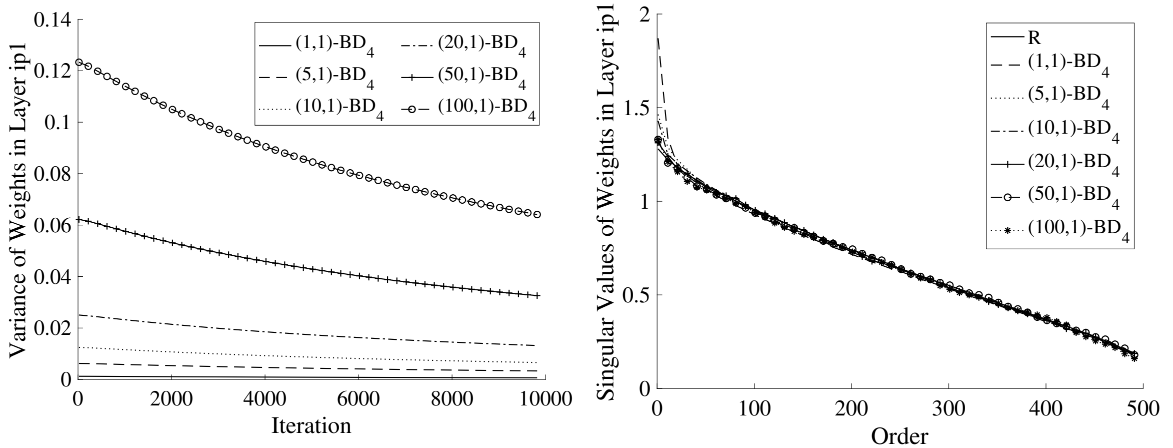

Figure 2.7: 10000 training iterations using LeNet-5 net on MNIST. (Left) Variance of weight matrix entries in layer ip1 in both the fully connected and block diagonal setting. (Right) Singular values of ip1 weight matricies for sister $(b_1, 1)$-BD$_4$ networks and of R, a random matrix with iid entries of equal dimension.

have the same variance as the entries in W_1. $\lambda_{(1/500)W_1W_1^\top}$ and $\lambda_{(1/500)RR^\top}$ align as I would expect from Figure 2.5 (Right).

If I make ip2 a BDIP layer as well in the LeNet-5 framework, the change in variance in ip1 sees minimal effect. For $b_1 = 1, 2, 5, 10, 50, 100$ and $b_2 = 1, 2, 5, 10$, the final variance in the ip1 layer weights using (b_1, b_2)-BD$_4$ over the final variance in the ip1 layer weights using $(b_1, 1)$-BD$_4$ is approximately 1 with error ≤ 0.05. [5]

In my experiments, the relationship between inner product layers in sister networks is independent of network architecture. I ran experiments on purely inner product layer networks without convolutional or other types of layers with the same results, and I will discuss a small purely inner product layer network briefly here. In a 3 layer network in which both hidden layers have 500 nodes and ReLu activation, I compare (1,1,1)-BD$_3$ to (1,100,1)-BD$_3$ on MNIST where in both cases ip2 is initialized with Xavier variance (Glorot and Y. Bengio 2010). After 10000 iterations, the ratio of the variance of the weight matrix entries in layer ip2 in block diagonal setting over the variance of the weight matrix entries in layer ip2 in the fully connected setting is 106.42.

Let $\sigma_{W_{b_2}}$ be the singular values of the ip2 layer weight matrix in the (1,b_2,1)-BD$_3$ architecture with only 3 inner product layers after 10000 iterations. The final variance of the ip2 layer weights in (1,1,1)-BD$_3$ is 0.0012. If R is a 500×500 truly random matrix with independent entries that have variance 0.0012, then $\mathrm{E}[|1 - \sigma_{W_1}/\sigma_R|] = 0.1162$. The final variance of the ip2 layer weights in (1,100,1)-BD$_3$ is 0.12. If R is a 500×500 truly random matrix with independent entries that have variance 0.12, then $\mathrm{E}[|1 - \sigma_{W_{100}}/\sigma_R|] = 0.0896$.

[5] The ip2 layer weights also *appear* to have the same behavior, but the matrix size is much smaller so the estimate of the variance is lower order and the asymptotic assumptions of Marchenko-Pastur are far from met.

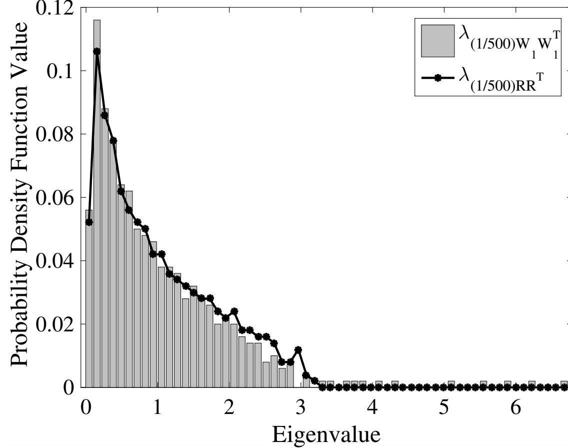

Figure 2.8: $\lambda_{\frac{1}{500}W_{b_1}W_{b_1}^\top}$ is the measured empirical spectral distribution of $\frac{1}{500}W_{b_1}W_{b_1}^\top$ where W_{b_1} is the ip1 layer weight matrix in the $(b_1, 1)$-BD architecture after 10000 training iterations on MNIST using LeNet-5. Bar graph of $\lambda_{\frac{1}{500}W_1W_1^\top}$ with plot of $\lambda_{\frac{1}{500}RR^\top}$ for a random matrix R with the same variance.

2.4.2 CIFAR10

In my experiments on CIFAR10 with Krizhevsky's Cuda-convnet (Krizhevsky 2012a), the first inner product layer weights are initialized using a Gaussian filler with standard deviation 0.1. Thus the ratio of variance of the weights in ip1 in the block diagonal case over that of the fully connected case at initialization is 1. Figure 2.9 (Left) indicates that this ratio is a good first order estimate for the final ratio in sister $(b_1, 1)$-BD$_5$ Cuda-convnet networks after 9000 iterations at which time the ratios are $1.02, 1.02, 1.01, 1.11, 1.21$ and 1.5 for $b_1 = 2, 4, 8, 16, 32$, and 64 respectively. Like with my experiments on MNIST, the relationship deteriorates as the number of blocks grows.

I compared the singular values of the weight matrix in layer ip1 for sister $(b_1, 1)$-BD$_5$ Cuda-convnet networks after 9000 training iterations to the singular values of a truly random 1024×64 matrix initialized with variance 7×10^{-3}, the final variance of the fully connected ip1 layer weights after training. I denote the random matrix R. As in the MNIST experiments, I aggregate the singular values of each block in the

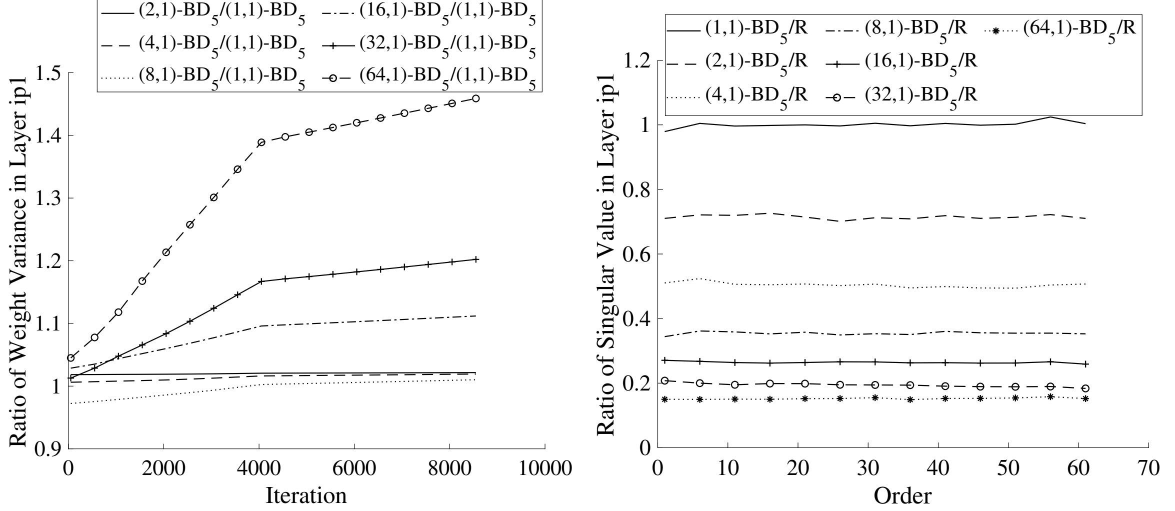

Figure 2.9: 9000 training iterations using Cuda-convnet on CIFAR10. (Left) Ratio of variance of ip1 weight matrix entries in block diagonal setting over variance of ip1 weight matrix entries in the fully connected setting. (Right) Ratio of trained ip1 weight matrix singular values over singular values of a truly random matrix with the same dimensions.

block diagonal method and sort them, and for an array of singular values arranged by order, division is done entry-wise. In Figure 2.9 (Right) I have written $(b_1, 1)$-BD$_5/R$ in the figure legend, but by this I mean entry-wise division of the ordered singular values of the ip1 weight matrix in $(b_1, 1)$-BD$_5$ Cuda-convnet over the ordered singular values of R. By convention, the lowest order singular values are the largest.

By the Marchenko-Pastur theorem, maintaining the variance of the entries in a random matrix while decreasing the matrix dimension by the some factor b_1 will alter the singular value distribution by a factor of $1/\sqrt{b_1}$; see equation (2.3). Figure 2.9 (Left) shows that the ratio of the ip1 initialization variance in $(b_1, 1)$-BD$_5$ Cuda-convnet over the ip1 initialization variance $(1, 1)$-BD$_4$ Cuda-convnet remains relatively constant after 9000 training iterations, and for b_1 blocks the ip1 layer weight matrix decreases in dimension by a factor of b_1. Figure 2.9 (Right) shows that the ratio of the singular values of the trained layer ip1 weight matrix in $(b_1, 1)$-BD$_5$ Cuda-convnet networks over the singular values of R is approximately $1/\sqrt{b_1}$ suggesting that

the trained layer ip1 weight matrix in $(b_1, 1)$-BD$_5$ Cuda-convnet networks behaves like a random matrix with deterioration as b_1 grows.

Again, the ratio of the variance and singular values of the ip1 layer weight matrix in block diagonal setting over that of the ip1 layer weight matrix in the fully connected setting after 9000 training iterations on CIFAR10 using Cuda-convnet best highlight the relationship to random matrix theory, but I also include the variance and the singular values without taking a ratio in Figure 2.10. The singular values of the fully connected ip1 layer weight matrix follow the curve that the singular values of the truly random matrix R create.

Figure 2.11 compares the probability density function of the singular values of a truly random matrix with independent entries to the measured distribution of the ip1 layer weight matrix singular values using the (1,1)-BD$_5$ architecture after 9000 training iterations. I use W_1 to denote the ip1 layer weight matrix, and R to denote a 1024×64 random matrix with iid entries that have the same variance the entries in W_1.

If in addition I make the ip2 a BDIP layer, again, the change in variance in ip1 sees minimal effect. For $b_1 = 1, 2, 4, 8, 16, 32, 64$, the final variance in the ip1 layer weights using the $(b_1, 2)$-BD$_5$ Cuda-convnet method over the final variance in the ip1 layer weights using the $(b_1, 1)$-BD$_5$ Cuda-convnet method is approximately 1 with error ≤ 0.03.

2.5 Discussion

I have shown that BDIP layers can reduce inner product layer size, training time and final execution time without significant harm to the network performance. I have also shown that random matrix theory gives informative results about relationships in network structure that are preserved through thousands of training iterations.

While traditional iterative pruning can reduce storage, the scattered surviving

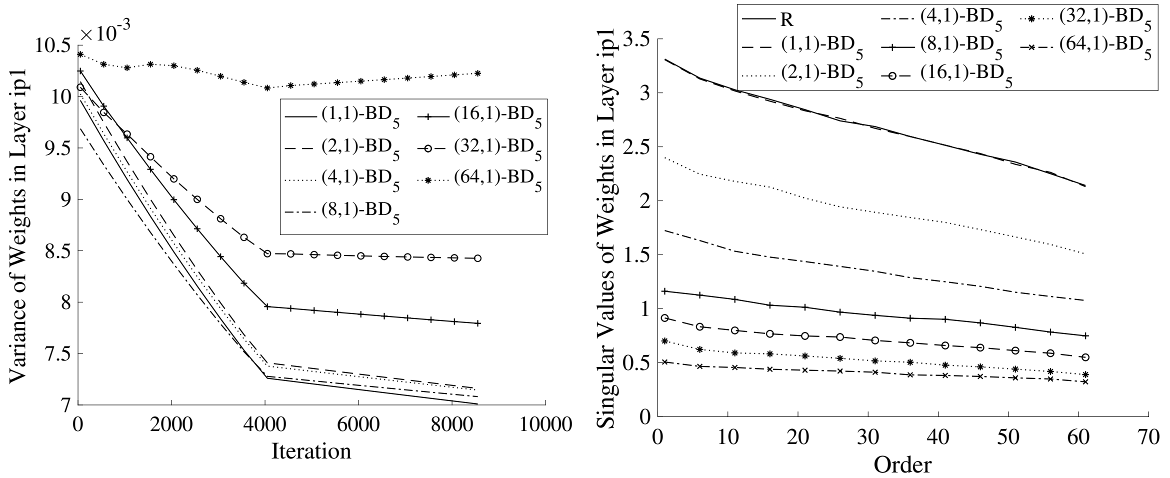

Figure 2.10: 9000 training iterations using Cuda-convnet on CIFAR10. (Left) Variance of weight matrix entries in layer ip1 in both the fully connected and block diagonal setting. (Right) Singular values of ip1 weight matricies for sister $(b_1, 1)$-BD$_5$ networks and of a random matrix with iid entries of equal dimension.

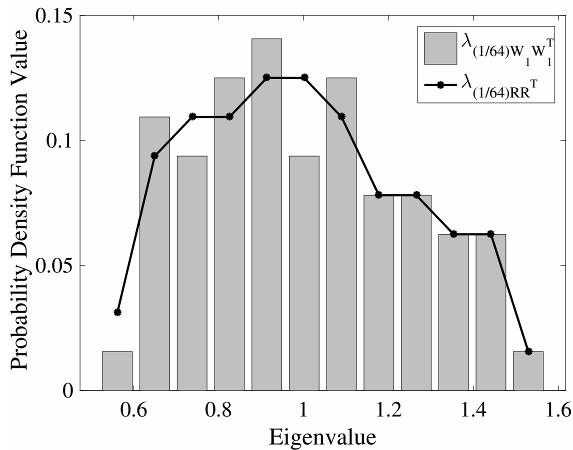

Figure 2.11: $\lambda_{\frac{1}{64}W_{b_1}W_{b_1}^\top}$ is the measured empirical spectral distribution of $\frac{1}{64}W_{b_1}W_{b_1}^\top$ where W_{b_1} is the ip1 layer weight matrix in the $(b_1, 1)$-BD$_5$ architecture after 9000 training iterations on the CIFAR10 dataset using Cuda-convnet framework. Bar graph of $\lambda_{\frac{1}{64}W_1W_1^\top}$ with plot of $\lambda_{\frac{1}{64}RR^\top}$ for a random matrix R with the same variance.

weights make sparse computation inefficient, slowing down both training and final execution time. My block diagonal methods address this inefficiency by confining dense regions to blocks along the diagonal. Without pruning, block diagonal method 1 allows for faster training time. IP-BDIP layers preserve the learning with focused, structured pruning that reduces computation for speedup during execution. In my experiments, IP-BDIP layers saw higher accuracy than the purely block diagonal method. The success of IP-BDIP layers supports the use of pruning as a mapping from large dense architectures to more efficient, smaller, dense architectures. Both methods make larger network architectures more feasible to train and use since they convert a fully connected layer into a collection of smaller inner product learners working jointly to form a stronger learner. In particular, GPU memory constraints become less constricting.

There is a lot of room for additional speedup with BDIP layers. Dependency between layers poses a noteworthy bottleneck in network parallelization. With structured sparsity like ours, one no longer needs a full barrier between layers. Additional speedup would be seen in software optimized to support weight matrices with organized sparse form, such as blocks, rather than being optimized for dense matrices. For example, for many small blocks, one can reach up to 6 fold speedup with specialized batched matrix multiplication (Jhurani and Mullowney 2015). Hardware has been developing to better support sparse operations. Block format may be especially suitable for training on evolving architectures such as neuromorphic systems. These systems, which are far more efficient than GPUs at simulating mammalian brains, have a pronounced 2-D structure and are ill-suited to large dense matrix calculations (Boahen 2014; Merolla et al. 2014).

I have established a connection between random matricies with independent entries and trained inner product layers; the group behavior resembles that of a random matrix with independent entries, but the individual weight updates have complex

dependencies. Similar random activity occurs in the mammalian brain and suggests looking at random matrix theory to support a network's plasticity or robustness. This connection could help evaluate network fitness. I have also shown that the relationship in structure between sister networks is perpetuated in the ratio of the change in variance after thousands of training iterations. I emphasize that this is a nontrivial relationship surviving various datasets, network architectures, and activation functions. Random matrix theory has been indispensable to the advancement of nuclear physics, quantum physics, neuroscience, and ecology, and has the potential to elevate artificial neural network analysis in the same manner.

CHAPTER III

Predictor-Corrector Gradient Descent

In this chapter, I propose a new training technique called Predictor-Corrector Gradient Descent (PCGD) that reduces the number of gradient descent iterations required to optimize any objective function that can be optimized using standard gradient descent. In PCGD I monitor the trends of the model parameters as the chosen model learns with gradient descent, and periodically adjust each parameter by inferring future values from the trend. A number of standard gradient descent iterations between predictions act to refine the predicted approximations. This alternating process works in much the same way that predictor-corrector methods for solving ordinary differential equations work.

While PCGD is a general adaptation to gradient descent, the benefits of using PCGD in place of traditional gradient descent are most prominent when learning a computationally costly model like a neural network. I will show that incorporating prediction into the training process of networks makes learning significantly more efficient. The human brain already utilizes predictions. Predictions are crucial to survival because they allow us to respond more appropriately to our surroundings and they improve reaction time. Perception is also impacted by brain predictions: our perceptions are a combination of expectations and sensory information (Heeger 2016; Luca and Rhodes 2016). Thus, if we wish to improve artificial neural network

efficiency, integrating prediction into training is a natural modification.[1]

3.1 Related Work

There is a plethora of work that supplements standard gradient descent in hopes of improving iterative training. Gradient noise and stale gradients have been successful adaptations to gradient descent (Ho et al. 2013; Neelakantan et al. 2015). Adapative Gradient techniques give frequently occurring features low learning rates and infrequent features high learning rates; these methods use the information theoretic idea that infrequent features carry more information about the data distribution (Dozat 2016; Duchi, Hazan, and Singer 2011; Kingma and Ba 2015; Tieleman and G. Hinton 2012; Zeiler 2012). Momentum and Nesterov's Accelerated Gradient (NAG) accumulate a descent direction across iterations to alleviate zig-zagging and accelerate convergence (Nesterov 1983; Polyak 1964) . There are also meta-learning methods that allow models to be trained jointly with their learning algorithm. Meta-methods may intelligently adjust hyperparameters like the learning rate, or learn the entire update term perhaps as a function of the batched gradient (Andrychowicz et al. 2016; Daniel, Taylor, and Nowozin 2016). Each of these techniques complement gradient descent to improve model learning and can be used in conjunction with my methods.

Prediction-correction methods are traditionally used in numerical analysis to integrate ordinary differential equations (Süli and Mayers 2003). Since their inception, predictor-corrector methods have been used in a variety of fields that require optimization like theoretical study of chemical reactions and time-varying convex optimization (Hratchian, Frisch, and Schlegel 2010; Simonetto et al. 2015). Prediction-correction has been incorporated into neural network training in the past by coevolving a pair of neural networks, a prediction network and a correction network (Y. Zhang, Chuang-

[1] One caution ought to be mentioned here: brain predictions also enable prejudices, so one must be careful how much trust is placed in predictions.

suwanich, et al. 2015; Y. Zhang, Yu, et al. 2015).

Scieur et al. propose a related learning algorithm to the one presented in this paper called Regularized Nonlinear Acceleration (RNA) (Scieur, d'Aspremont, and Bach 2016). RNA computes estimates of the optimum from a nonlinear average of a history of iterations produced by an optimization method like gradient descent. Like in RNA, the prediction step in PCGD is based on a history of parameter values obtained with gradient descent. However, my predictions use parameter specific linear regression rather than a nonlinear average of complete historical iterations. Making parameter specific predictions with linear regression allows my method to update predictions incrementally, which removes the need to keep all historical iterations relevant to a particular prediction. RNA must store the entire iteration history relevant to a particular prediction, which makes this method unfeasible for training large neural networks.

3.2 Methodology

PCGD uses best fit predictions and stochastic gradient descent in tandem. When estimating the trend in the model parameters through training, I will use fit functions for which the least squares problem has a closed form solution using the normal equations. One could use more complex fit functions, but I want to avoid needing an extra iterative process. Using only least squares problems with closed form solutions to make parameter predictions also saves memory because they can be solved incrementally, avoiding the need to store a long history of model snapshots; this is particularly useful when learning a training a parameter-heavy model like a deep neural network.

I will define the algorithm around the gradient descent iterations. I will make parameter predictions every p gradient descent iterations and collect snapshots of the model parameters every s^{th} gradient descent iteration where $p > s$ and $s|p$. Parameter

predictions only consider the previous p/s model snapshots. Since $p > s$, only a sparse history of snapshots are considered. We'll call p the prediction increment and s the snapshot increment. For the remainder of this paper, the variables p and s will retain this definition.

Suppose our model has n parameters. Let $f(\boldsymbol{a}, x) : \mathbb{R}^c \times \mathbb{R} \to \mathbb{R}$ be our chosen fit function class for parameter prediction. For each model parameter, θ, I aim to solve for $\boldsymbol{a}$, such that $f(\boldsymbol{a}, x)$ estimates a future value of θ for a chosen prediction length x. $f(\boldsymbol{a}, x)$ has c unknowns where $c \leq p/s$. Define $F(A, x) : \mathbb{R}^{c \times n} \times \mathbb{R} \to \mathbb{R}^n$ such that the ith entry of $F(A, x)$ is $f(\boldsymbol{a}_i, x)$ where $\boldsymbol{a}_i$ is the ith column of A. When using PCGD, model parameter vector $\boldsymbol{\theta} \in \mathbb{R}^n$ receives the update,

$$\begin{aligned}
\boldsymbol{v}_t &= -\epsilon \nabla L(\boldsymbol{\theta}_t) \\
\boldsymbol{\theta}_{t+1} &= \begin{cases} F(A_{t+1}, l_{t+1}) & \text{if } \ t+1 \equiv 0 \mod p \\ \boldsymbol{\theta}_t + \boldsymbol{v}_t & \text{otherwise} \end{cases}
\end{aligned} \tag{3.1}$$

where L is the desired loss function, ϵ is some learning rate, $l_{t+1} \geq p/s$ is an increasing prediction length and $A_{t+1} \in \mathbb{R}^{c \times n}$, minimizes the L_2-norms of the columns of $JA_{t+1} - \Theta_{t+1}$. Here, $J \in \mathbb{R}^{(p/s) \times c}$ has entries $J_{i,j} = \partial f(\boldsymbol{a}, i)/\partial a_j$, and the ith row of Θ_{t+1} is the vector $\boldsymbol{\theta}_{t+1-p+is}^\top$ for $i < p/s$ and $\boldsymbol{\theta}_t^\top + \boldsymbol{v}_t^\top$ for $i = p/s$.[2] Note that the columns of A_{t+1} each solve independent least squares problems for particular model parameters; the systems are overdetermined if $c < p/s$. I use one fit function class, f, but calculate model-parameter specific fit function variables. One could easily add regularizers or momentum to the velocity term, $\boldsymbol{v}_t$. l_{t+1} is an increasing prediction length dependent on the gradient descent iteration, but one could also consider an adaptive, or parameter specific prediction length. Iterations, t, in which $t \equiv 0 \mod p$ constitute the 'predictive' step in PCGD, and all other gradient descent iterations

[2] Note that the jacobian, J, is not specific to the column of A_{t+1} .

comprise the 'corrective' step.

I solve for prediction fit function variables A_{t+1} incrementally so as to minimize the extra storage required to perform PCGD. Fit function variables are updated at snapshot intervals. Let $\Theta_{t+1}^{(i)}$ denote the shorter matrix containing only the first i rows of Θ_{t+1}. Similarly, $J^{(i)}$ is the shorter matrix containing only the first i rows of J. When c snapshots have been recorded, I solve $J^{(c)}A_{t+1} = \Theta_{t+1}^{(c)}$ for the fit function variable matrix A_{t+1}; with c snapshots $J^{(c)}A_{t+1} = \Theta_{t+1}^{(c)}$ is a determined system. After this initial solve, only A_{t+1} must still be stored, $\Theta_{t+1}^{(c)}$ is no longer needed. At snapshot intervals $c+1$ through p/s I update the fit function variable matrix using the incremental least squares algorithm found in (Cassioli et al. 2013). That is, for $i \in [c+1, p/s]$, I update,

$$A_{t+1} \leftarrow A_{t+1} + \boldsymbol{y}_i \left(\left(\boldsymbol{\theta}_{t+1}^{(i)} \right)^\top - \boldsymbol{j}_i^\top A_{t+1} \right) \tag{3.2}$$

where $\left(\boldsymbol{\theta}_{t+1}^{(i)} \right)^\top$ is the ith row in Θ_{t+1}, $\boldsymbol{j}_i^\top$ is the ith row of J, and $\boldsymbol{y}_i$ is the solution to $\left(J^{(i)}\right)^\top J^{(i)} \boldsymbol{y}_i = \boldsymbol{j}_i$.

This process then repeats writing over old fit function variables and parameter history in memory. Since fit functions variables are parameter specific, they can be updated layer-wise. If a model has n total parameters, PCGD requires storing at most an additional $O(cn)$ values in memory at any one time during training when using a fit function with c unknowns. The size of the extra storage is c times the size of layers not being currently being updated plus at most $2c$ times the size of the layer currently being updated.

By using an incremental least squares approach and solving for parameter specific best fit functions, I am able to conserve memory during training; without this approach one would need to store np/s parameter history values. This makes PCGD a feasible technique for training large neural networks provided c is small. Given the

same history, RNA would solve for p/s coefficients for p/s entire network snapshots to obtain a nonlinear average of the whole snapshots (Scieur, d'Aspremont, and Bach 2016). Hence, RNA would require storing all np/s parameter history values. However, for the memory conservation afforded by incrementally updating fix functions, one pays a little extra work. Rather than solving for A_{t+1} directly, one must perform $p/s - c + 1$ incremental updates to A_{t+1}.

3.3 Relationship to Nesterov's Accelerated Gradient

One could make predictions every iteration, which would bring my method closer to some existing accelerated gradient schemes. If one made predictions every iteration using a linear fit function my algorithm could be written,

$$\begin{aligned} \boldsymbol{z}_t =& \begin{cases} \boldsymbol{\theta}_t & \text{if} \quad t < p \\ A_t^\top \begin{bmatrix} 1 & l_t \end{bmatrix}^\top & \text{otherwise} \end{cases} \\ \boldsymbol{\theta}_{t+1} =& \boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t) \end{aligned}$$

where A_t minimizes the L_2-norms of the columns of $JA_t - \Theta_t$. Here, $J \in \mathbb{R}^{(p/s)\times 2}$ has $\begin{bmatrix} 1^{i-1} & 2^{i-1} & \cdots & (p/s)^{i-1} \end{bmatrix}^\top$ for its ith column vector, and $\Theta_t \in \mathbb{R}^{(p/s)\times n}$ has $\boldsymbol{\theta}_{t-p+is}^\top$ for its ith row vector. With $p = 2$ and $s = 1$, this begins to look quite a bit like NAG algorithm which makes the update,

$$
\begin{aligned}
\boldsymbol{z}_t =& (1-\gamma_{t-1})\boldsymbol{\theta}_t + \gamma_{t-1}\boldsymbol{\theta}_{t-1} \qquad \text{with } \boldsymbol{z}_0 = \boldsymbol{\theta}_0 \\
\boldsymbol{\theta}_{t+1} =& \boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t)
\end{aligned}
$$

for specifically chosen series $\{\gamma_t\}_{t=0}^{\infty}$. With $l_t = 2 - \gamma_{t-1}$ these methods are identical. For continuously differentiable, smooth, convex loss functions NAG can achieve a global convergence rate of $O(1/t^2)$ (Beck and Teboulle 2009; Nesterov 1983). A natural extension of NAG incorporates a history of three points such that the update is

$$
\begin{aligned}
\lambda_t =& \left(1 + \sqrt{1 + 4\lambda_{t-r}^2}\right) \Big/ 2 \\
\boldsymbol{z}_t =& \begin{cases} \frac{\lambda_{t-1}}{\lambda_t}\boldsymbol{\theta}_t + \frac{(\lambda_t - 1)}{\lambda_t}\boldsymbol{\theta}_{t-r+1} - \frac{(\lambda_{t-1}-1)}{\lambda_t}\boldsymbol{\theta}_{t-r} & \text{if } t > r \\ \boldsymbol{\theta}_t & \text{otherwise} \end{cases} \\
\boldsymbol{\theta}_{t+1} =& \boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t)
\end{aligned} \tag{3.3}
$$

where $\lambda_0, \cdots, \lambda_{r-1} = 0$ and $r \in \mathbb{Z}^{>0}$.

Theorem 3.3.1. *Let L be a convex, continuously differentiable and β-smooth function that admits a minimizer $\boldsymbol{\theta}^* \in \mathbb{R}^n$. Given an arbitrary initialization $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, for $T > r$ and $\epsilon = 1/\beta$, update scheme (3.3) satisfies,*

$$
\sum_{t=T-r}^{T} \lfloor (t+1)/r \rfloor^2 \left(L(\boldsymbol{\theta}_{t+1}) - L(\boldsymbol{\theta}^*)\right) \leq 2\beta \|\boldsymbol{z}_r - \boldsymbol{\theta}^*\|_2^2 \, .
$$

When $r = 1$ this reduces to NAG. If in addition we assume strong convexity of our objective function L the convergence rate becomes clearer.

Corollary 3.3.1.1. *Let L be strongly convex with parameter $m > 0$, continuously differentiable and β-smooth function that admits a minimizer $\boldsymbol{\theta}^* \in \mathbb{R}^n$. Given an arbitrary initialization $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, for $T > r$ and $\epsilon = 1/\beta$, update scheme (3.3) satisfies,*

$$\sum_{t=T-r}^{T} \lfloor (t+1)/r \rfloor^2 \left(L(\boldsymbol{\theta}_{t+1}) - L(\boldsymbol{\theta}^*) \right) \leq \frac{\beta^2 \|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2^2}{mr} \,.$$

The order of r in the denominator on each side of the above inequality is the same. Hence, for $m = \beta$, $\min_{t\in\{T-r,\cdots,T\}}\{L(\boldsymbol{\theta}_{t+1}) - L(\boldsymbol{\theta}^*)\}$ converges at the same rate as NAG. The proof of Theorem 3.3.1 and Corollary 3.3.1.1 can be found in Appendix C.

In this well-behaved, theoretical environment, updating based on a linear combination of older values maintains the convergence rate of NAG. However, update method (3.3) is not practical for deep learning because it requires $r\times$ the memory to save a history of network parameter values. Instead, making parameter predictions every p^{th} iteration, as in update method (3.1), makes the additional memory requirement significantly more practical. In the setting of neural network parameters, update method (3.1) has the capacity to outperform NAG. Considering an evenly distributed history of values extending further in the past allows one to de-noise trends. By incorporating a longer history, method (3.1) can afford to make predictions further into the future while minimizing additional memory requirements.

In comparison to NAG, employing update scheme (3.1) requires more memory for the fit function variables A_t, but performs less work as snapshot increment s and prediction increment p increase since fit function updates and parameter predictions happen less often. One must strike a balance though: for large p and large p/s one should be able to predict model parameters with more confidence provided the chosen

fit function is well suited for the trend, but large p will exhibit delayed performance. Method (3.1) introduces a number of new hyperparameters that can be tuned for a particular task.

3.4 Experimental Results

The goal of my approach is to decrease the number of training epochs needed for a neural network to reach a particular testing accuracy. To test this, I ran experiments on the SVHN (Netzer et al. 2011), and CIFAR10 (Krizhevsky 2009) datasets using Krizhevsky's cuda-convnet with 4 hidden layers (Krizhevsky 2012a). This net does not produce state-of-art accuracies for these datasets, but rather highlights the improvement seen by PCGD when compared to SGD. I implement my work in Caffe, which provides this architecture in their CIFAR10 "quick" example. I trained using batch size 100. Unless otherwise specified, hyperparameters and initialization distributions provided by Caffe's "quick" architecture are left unchanged (see Appendix B for an architecture outline and hyperparameter details). All experiments are run on the Bridges' NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center. Training is done with batched gradient descent using the cross-entropy loss function on the softmax of the output layer.

In this paper I will only use linear fit functions to make parameter predictions. That is, the fit function class is $f(\boldsymbol{a}, x) = a_1 + a_2 x$ and the number of fit function variables to solve for for each network parameter is $c = 2$. In this case, a network with n parameters requires storing an additional $2n$ values. If m is the maximum number of iterations I will train, p is the prediction increment and s is the snapshot increment, define $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, t) = b_1 + b_2 \left(t/p\right)^d$ where and $\boldsymbol{b}$ is chosen such that $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, 0) = p/s + u_1$ and $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, m) = p/s + u_2$ for some $u_1, u_2 \in [0, 2p/s]$, $u_1 < u_2$. I chose my prediction length such that $l_t = g_{(d,\boldsymbol{u})}(\boldsymbol{b}, t)$. This means that at iteration p, PCGD tries to predict what the network weights will be at iteration $p + su_1$ and sets the weights to

those predicted values. Similarly, at iteration m, PCGD *would* try to predict what the network weights would be at iteration $m + su_2$, but I do not make the last, or last few, predictions because immediately after predicting there is often a slight drop in accuracy that needs to be corrected by some gradient descent steps. This slight drop after predicting could be minimized by less aggressive predictions or better fit function choices, but I chose to simply leave out the last few predictions. It is a good idea to have u_1 small because parameter trends can alter and we do not want to be over-influenced by start-up trends.[1]

I will compare PCGD with NAG and SGD. I also consider a hybrid method combining NAG and PCGD, abbreviated as NAG-PCGD. To combine the two methods I nest NAG updates inside PCGD updates; the update scheme for NAG-PCGD is written out explicitly in Appendix D. When training with PCGD and NAG-PCGD, I use prediction increment $p = 150$, snapshot increment $s = 15$ for all of my experiments. When plotting accuracy results, I will plot the maximum testing accuracy seen so far by that training iteration against iterations. While training, testing accuracy is usually noisy, which can obscure differences in performance when comparing different methods. Plotting the maximum testing accuracy seen so far displays these differences more clearly. There was no noticeable difference in the amount of noise seen in the testing accuracy for the various methods in my experiments.

3.4.1 SVHN

I experimented on the SVHN dataset with Krizhevsky's cuda-convnet (Krizhevsky 2012a). The base learning rate was 0.001 and dropped by a factor of 10 after 4,000 iterations (see Appendix B for an architecture outline and hyperparameter details). Testing took place every 50 training iterations. When training with PCGD and NAG-PCGD, I use prediction length $l_t = g_{(6,[5,10])}(\boldsymbol{b}, t)$.

Figure 3.1 (Left) plots the maximum accuracy seen so far against iterations using

standard SGD, NAG, PCGD and NAG-PCGD. Figure 3.1 (Right) plots the slopes of the curves in Figure 3.1 (Left) versus iteration. I show the iterations of steepest accuracy increase to highlight the difference in convergence rates of the various methods. NAG and NAG-PCGD initially increase at nearly the same rate which is $\approx 4\times$ faster than PCGD and SGD. Around iteration 450 PCGD leaves behind SGD, begins to catch up to NAG and eventually supersedes it. NAG-PCGD tends to hug the top of all the other curves exhibiting the benefits of both sub-methods. Confined to 2000 iterations, NAG-PCGD gives the best results. At iterations 4000 when the learning rate decreases by a factor of 10, there is another jump in accuracy where we can see the difference in convergence rates again on a smaller scale.

After 9000 iterations, the network trained using traditional SGD achieves a final accuracy of 91.96%, NAG has a final accuracy of 92.38%, PCGD has a final accuracy of 92.42%, and NAG-PCGD has a final accuracy of 92.34%. SGD hit a maximum testing accuracy of 92.06% at iteration 8600, NAG took 4700 iterations to reach this accuracy level, PCGD also took 4700 iterations and NAG-PCGD took 5100 iterations. That is, PCGD reached SGD's testing maximum in just over half the number of training iterations that SGD took.

3.4.2 CIFAR10

I also trained Krizhevsky's cuda-convnet on the CIFAR10 for 195,000 iterations (see Appendix B for an architecture outline and hyperparameter values for hyperparameters not overwritten here). The base learning rate was 0.001. I dropped the learning rate by a factor of 10 after 60,000 iterations and again after 125,000 iterations. Testing took place every 250 training iterations. I used $l_t = g_{(4,[5,10])}(\boldsymbol{b}, t)$ for my prediction length at prediction intervals.

Figure 3.2 (Left) shows maximum accuracy results through training using SGD, ADAM, NAG, PCGD and NAG-PCGD. Again, I show only the iterations of steepest

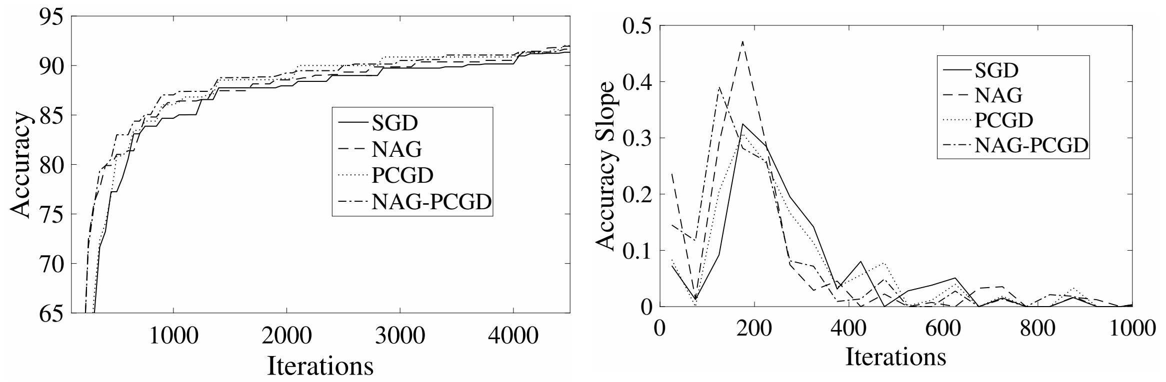

Figure 3.1: (Left) Maximum accuracy results on the SVHN data set. Testing takes place every 50 training iterations (Right) Slope of Left Figure versus iterations.

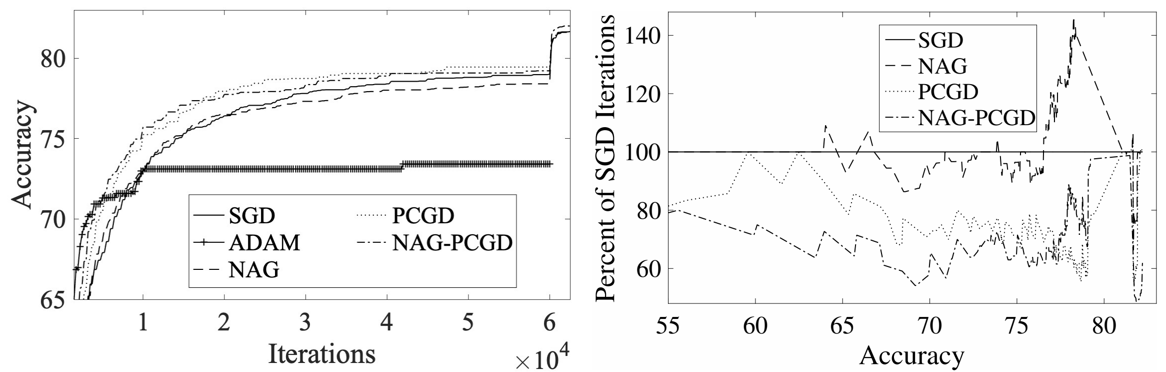

Figure 3.2: Results on the CIFAR10 data set. (Left) Maximum Accuracy versus iterations. Testing takes place every 250 training iterations. (Right) Percent of SGD iterations each method took to reach a particular accuracy.

Table 3.1: Results on the CIFAR10 data set. Percent of SGD iterations each method took to reach a particular accuracy.

Accuracy	**SGD**	**NAG**	**PCGD**	**NAG-PCGD**
65%	100%	92.86%	78.57%	64.29%
70%	100%	92.31%	73.07%	65.38%
75%	100%	96.42%	71.42%	71.42%
81.7%	100%	73%	56%	50%

accuracy increase. I only ran ADAM for 60000 iterations as it was not performing well compared to other methods. I used $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-/8}$ when running ADAM as suggested in (Kingma and Ba 2015). Here, the testing increment is larger than my prediction increment which may hide any initial convergence advantage of NAG over PCGD. Given more time to excel, PCGD shows performance advantages over NAG; NAG does not even consistently outperform SGD per iteration. At any one time, NAG is at most 3.18% more accurate than SGD, PCGD is at most 3.91% more accurate than SGD, and NAG-PCGD is at most 6.49% more accurate than SGD.

Figure 3.2 (Right) shows, for a given accuracy, the percent of SGD iterations each method took to reach that accuracy. Table 3.1 highlights some values from Figure 3.2 (Right). That is, if it took SGD x iterations to reach a particular accuracy for the first time, and PCGD took y iterations to reach that accuracy for the first time, then the value plotted for PCGD at that accuracy is $100 \times y/x$. This figure shows PCGD generally reaching particular accuracies before SGD and NAG-PCGD generally reaching accuracies before PCGD. SGD took 114,000 iterations to become 81.7% accurate. Training with NAG yielded 81.7% accuracy in 73% of the iterations required by SGD to reach this accuracy, training with PCGD yielded 81.7% accuracy in 56% of the iterations required by SGD and training with NAG-PCGD yielded 81.7% accuracy in 50% of the iterations required by SGD. That is, PCGD took only 77% of the iterations required by NAG to reach 81.7% accuracy.

For these values of s and p, using PCGD does not noticeably increase the average iteration runtime when compared with SGD. For both methods, the average forward-backward pass took $\approx$ 46ms when using batch size 100 on Bridges' NVIDIA P100 GPU; time was measured using caffe time benchmarks.

3.5 Discussion

I have developed a general adaptation to gradient descent and considered the impact in the case of training neural networks. Predictor-Corrector Gradient Descent reduces the number of iterations required to learn by incorporating traditional predictor-corrector inspired ideas into classic gradient descent.

I have shown that PCGD can significantly decreases the number of training epochs needed for a network to reach a particular testing accuracy when compared to stochastic gradient descent. On both datasets considered, PCGD reduced the number of required iterations to reach SGD maximum accuracy by nearly one half. When two identical networks are allowed to train for the same number of iterations, the networks trained using PCGD regularly outperforms the network trained using SGD. I have also shown that PCGD can outperform Nesterov's Accelerated Gradient for more complex learning problems requiring more training. By substantially reducing the number of iterations required to reach a particular accuracy, PCGD can make training large networks more feasible in cases where one can afford to increase the training storage by a small constant multiple.

I have also considered the theoretical case of a strongly convex, continuously differentiable and smooth objective function and showed that updating parameters as a linear combination of historical values preserves the convergence rate of NAG. Although my experimental environment is far from this hypothetical one, this theory holds true when using PCGD to train neural networks. After an initial delay, I found PCGD can outperform NAG.

In this work, I only used linear fit functions and a single prediction length for every network parameter. These choices worked well, but there is room for additional exploration. One may see further improvement by using a dynamic value for the prediction interval p.

CHAPTER IV

Generating Artificial Core Users for Interpretable Condensed Data

Recommendations Systems improve the user experience by providing personalized, curated recommendations in a large and complex information space. Collaborative Filtering methods exploit community data to uncover correlations between users and items that can be used to make recommendations. Within collaborative filtering, latent factor models are considered state-of-the-art; these models approximate highly redundant data with low rank matrix decompositions (Aggarwal 2016; Hu, Koren, and Volinsky 2008; Koren 2008; Koren, Bell, and Volinsky 2009; Shi, Larson, and Hanjalic 2014).

Unfortunately, many matrix factorizations methods are transductive and cannot be leveraged to make predictions for users outside of the original training set (Aggarwal 2016). Orthonormal matrix factors can be used more easily to make out-of-sample predictions because the matrix factors capture geometric traits of the item and user spaces. However, achieving orthonormality of the matrix factors can be expensive for large datasets.

The majority of research in recommender systems focuses on developing new collaborative filtering and hybrid methods, which are often highly optimized for a specific application. There is a smaller body of research looking at identifying the most useful

users who carry most of the relevant information, and separating out those users as Core Users for making recommendations. With a smaller core set of users, making out-of-sample predictions becomes a reasonable task. Reducing a dataset size without much loss of information improves recommendation efficiency as well as storage costs; numerous fields outside of recommender systems would benefit from this ability.

However, available Core User methods have limited representation ability, in that they are selected from existing user data. In other words, the recommendation success of Core Users is bounded by quality of user data available. Improving the recommendation accuracy of Core Users makes data abstraction more effective for applications in data augmentation, bots mimicking population behavior, data mining, privacy, statistics and many more. In this chapter, I develop a method of generating Artificial Core Users (ACUs) that improves the recommendation accuracy of real Core Users. I combine latent factor models, ensemble boosting and K-means clustering, to generate a small set of Artificial Core Users (ACUs) from real Core User data. My ACUs incur a small amount of additional memory storage when compared to real Core Users, but remain a reduction in memory storage compared to the original dataset. Artificial Core Users improve the recommendation accuracy of real Core Users while remaining good centroids for the complete recommendation dataset. Since ACUs act as good centroids for the complete dataset, ACUs blend in well with the real dataset even though they are generated artificially. But unlike real Core Users, ACUs have complete ratings on all items, providing more immediately interpretable information to scientists.

4.1 Related Work

The inductive matrix completion problem assumes that a ratings matrix is generated by applying feature vectors to a known low-rank matrix (Jain and Dhillon 2013; Xiao Zhang, Du, and Gu 2018). This is relevant for making out-of-sample recom-

mendations if we assume that the latent factors contain some ground truth about the dataset that applies to out-of-sample users. As mentioned above, latent factors produced by most methods are sadly transductive (Aggarwal 2016). To combat this, some have worked on improving the efficiency of using a singular value decomposition in recommender systems (Sarwar† et al. 2002).

Using clustering is one of the earliest attempts to decrease the number of users you need to work with in order to make recommendations (Aggarwal 2016); rating predictions can be made using only information from the relevant cluster. Alternatively, Zeng et al. 2014 study the relevance of different users and find that there exists an "information core" made up of some key users. They found that the number of the Core Users is around 20 percent of the entire dataset, and that the recommendation accuracy produced by only relying on the Core Users can reach 90 percent of that produced using every user in the dataset. Zeng et al. 2014 use a generalized K-nearest Neighbor algorithm using various relevancy metrics to measure 'nearness'. They ran experiments using degree-based, resource-based and similarity-based measures, to select their Core Users; all of these measures are graphical in nature. Since this work a few other methods for selecting Core Users have emerged. Z. Li, Lei, and Shuyan 2016 use a long-tail-distribution-based measure to select their core uses. Cao and Kuang 2016 introduced a new measure to identify Core Users based on trust relationships and interest similarity; this work extends beyond graphical knowledge to include the semantic meaning of items. Kuang, Cao, and Chen 2017 use a combination of the measures proposed in (Cao and Kuang 2016) and (Z. Li, Lei, and Shuyan 2016).

Recently, deep learning has made an appearance in recommender systems either in the form of integration models or neural network models (S. Zhang, Yao, and A. Sun 2017). Integration models use neural networks to uncover disguised features in auxiliary information, like item descriptions (H. Wang, N. Wang, and Yeung 2015; H. Wang, Xingjian, and Yeung 2016), user profiles (S. Li, Kawale, and Fu 2015) and

knowledge bases (F. Zhang et al. 2016). The uncovered features are then incorporated into a collaborative filtering framework to produce hybrid recommendations. Neural network models on the other hand perform collaborative filtering directly via modeling the interaction function between users and items (X. He et al. 2017; Q. Li, Zheng, and X. Wu 2018; Sedhain et al. 2015; Strub, Mary, and Gaudel 2016; Y. Wu et al. 2016). Using deep learning to make recommendations means the models are better equipped to recognize nonlinear relationships in the data, but it also means that the models inherit all the training difficulties neural networks face compounded by the difficulties of working with sparse data.

There is a small body of work that injects fake users into a recommendation system either for adversarial goals (Christakopoulou and Banerjee 2018; Lam and Riedl 2004; OMahony, Hurley, and Silvestre 2002) or utilitarian data augmentation goals (Sarwar et al. 1998). Typically, the fake users are hand-coded, but Christakopoulou and Banerjee 2018) create adversarial fake user profiles for a recommendation system using generative adversarial nets. Sarwar et al. 1998 on the other hand used simple content filtering bots to generate a few users with dense ratings to improve the recommendations. Tackling the new user, or cold start, problem is an issue in recommender systems that has brought about some interesting work on determining which item preferences and item features are most informative (Rashid, Albert, et al. 2002; Rashid, Karypis, and Riedl 2008; Seroussi, Bohnert, and Zukerman 2011).

4.2 Methodology

I propose an offline technique to generate a small set of Artificial Core Users (ACUs) who's dense ratings matrix can be stored in condensed, low order form and used to make recommendations for out-of-sample users without much additional work. This method uses latent factor models, ensemble boosting and K-means clustering to learn a small group of artificial users who's feature vectors capture correlations in

the full dataset. This is a collaborative filtering method that uses an existing sparse ratings matrix for a set of users and items.

The algorithm requires a set of training users represented by their ratings on a given set of items, which will be used to update ACU ratings during learning. To measure the abstraction of the ACUs, I also have a set of testing users that does not overlap with the training user set from the same dataset. Let R be an $m \times n$ sparse ratings matrix for m training users and n items.[1] The ACU set size will be small relative to the size of the dataset; if s is the number of ACUs, I pick s such that $s \ll m$. Let R_{ACU} denote the $s \times n$ ratings matrix for my ACUs. There are a number of ways one could initialize the ACU ratings; in my work, ACU ratings will be initialized from Core User ratings. [2]

The training algorithm is given in Algorithm 1. I trained row blocks of R_{ACU} together using batches of training users. By learning ACUs in blocks, I incorporate boosting results and reduce our workload (Breiman 1998; Schapire 1990). Algorithm 1 learns orthonormal decompositions for each block, leveraging the relevance of orthonormal decompositions to out-of-sample relationships.[3] Let $R_{ACU}^{(b)}$ denote the bth row block in R_{ACU}, and R_i denote the sparse ratings matrix of the ith batch of training users. For simplicity, I assume that each block of R_{ACU} has the same number of rows and similarly each training batch has the same number of users. I divide R_{ACU} into ζ equal sized row blocks such that for $b \in \{0, \ldots, \zeta\}$ $R_{ACU}^{(b)}$ is a $\left(\frac{s}{\zeta} \times n\right)$ matrix. Let I denote the training batch index set such that for $i \in I$, R_i is a $\left(\frac{m}{|I|} \times n\right)$ matrix. The variables α and β are learning rates, while the variables λ and γ are regularization coefficients. Lines 15 and 20 in Algorithm 1 are derived from stochastic gradient descent algorithms. Line 18 updates the rows of U_{ACU} as the $\frac{s}{\zeta}$-means of $\frac{m}{|I|}$ row vectors in U_i,

[1] I did choose to normalize the variance and mean center the rows of R before proceeding; any adjustments made here can be accounted for at recommendation time.

[2] As with any learning process, one can see a lot of improvement by selecting the right initialization. There is room to try supplementing Core User ratings to improve the initialization of R_{ACU}.

[3] Computing the singular value decomposition of a large matrix is known to be time consuming, so learning in blocks saves quite a bit of time.

which is described in detail in Algorithm 3. Embedding Algorithm 3 inside Algorithm 1 effectively trains the Artificial Core User user-space matrix, U_{ACU}, with Mini-Batch K-means (Sculley 2010). By incorporating Algorithm 3 into the learning process we expect to keep the resulting Artificial Core User ratings matrix interpretable. That is, we expect the resulting Artificial Core User ratings to resemble that of real users, but provide complete rating information in place of sparse data. As with any learning method, the amount of regularization will be problem dependent; one may find that they can get away with $\lambda, \gamma = 0$ because lines 16 and 23-24 in Algorithm 1 have a regularizing effect. Lines 23-24 ensure that we don't stray to far from an orthonormal decomposition during learning. The while loop on line 10 takes a simplistic approach to the inductive matrix completion problem given a set of features; this subprocess could likely be improved by existing work (Jain and Dhillon 2013; Xiao Zhang, Du, and Gu 2018).

Algorithm 1 should be run offline to produce a set of Artificial Core Users. After which, one can use $V_{ACU}S_{ACU}$ to make faster recommendations for out-of-sample users. One may also be interested in using R_{ACU} as a condensed version of their dataset.

This process resembles the way one would train a neural network and shares similarities with neural network models (X. He et al. 2017; Q. Li, Zheng, and X. Wu 2018; Sedhain et al. 2015; Strub, Mary, and Gaudel 2016; Y. Wu et al. 2016). Neural network models generally use autoencoders. One layer Neural Collaborative Autoencoders with no output activation can be reformulated to parallel matrix factorization (Q. Li, Zheng, and X. Wu 2018). Unlike Algorithm 1, existing neural network models are not concerned with interpretability.

I evaluate my work using two metrics: item vectors testing error and sparse K-means error. To measure the item vectors testing error, I use Algorithm 2 on an independent set of testing users. Algorithm 2 is a simplistic recommender model

that tries to predict missing user ratings. It decomposes the row blocks of R_{ACU} into orthogonal components using a singular value decomposition. Then, for a set of real testing users, Algorithm 2 optimizes a set of unit vectors to accompany the item vectors extracted from the decomposition of R_{ACU} - there is no new learning done in the item vectors. In this way, Algorithm 2 evaluates the generality of the item vectors extracted from R_{ACU} and the potential for high quality recommendations using a more elaborate learning model. It should be noted that, as a simplistic recommender model, Algorithm 2 produces far from state-of-the-art recommendation results, but is useful for our purposes of evaluating whether Artificial Core Users better capture the information in a recommendation dataset than real Core Users do. To compute the sparse K-means error of R_{ACU} I make a few modifications to Algorithm 3 using a sparse first input matrix; this results in Algorithm 4. Algorithm 4 estimates how well the Artificial Core Users represent an average collection of real users, R_{ACU}, by measuring how well the ACUs serve as centroids for our real testing users.

Algorithm 1 Generate_ACUs(R)

1: Let m be the number of users (row dimension of R) and n be the items (column dimension of R).
2: Initialize constants α, λ, β, γ. Initialize matrix R_{ACU} with dimensions $s \times n$.
3: **while** not done **do**
4:     **for** $b = 0$ to ζ **do**
5:         Find orthonomal $\left(\frac{s}{\zeta} \times k\right)$ matrix $U_{ACU}^{(b)}$, orthonomal $(n \times k)$ matrix $V_{ACU}^{(b)}$ and diagonal $(k \times k)$ matrix $S_{ACU}^{(b)}$ such that $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^{\top} \approx R_{ACU}^{(b)}$ and $k \leq \min\left(\frac{s}{\zeta}, n\right)$.
6:         Set $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$.
7:         Select i randomly from training user batch index set I.
8:         Initialize U_i.
9:         $j = 0$.
10:         **while** not done **do**
11:             Compute sparse $E = R_i - U_i \left(V_{ACU}^{(b)}\right)^{\top}$ for the specified entries of R_i. Other entries of E remain zero.
12:             **if** not done **then**
13:                 **if** $j \mod 2 = 0$ **then**
14:                     $U_i \leftarrow (1 - \beta\gamma) U_i + \beta E V_{ACU}^{(b)}$.
15:                     Normalize the columns of U_i.
16:                 **else**
17:                     **if** $j \mod 4 = 1$ **then**
18:                         Means_Update$\left(U_i, U_{ACU}^{(b)}\right)$.
19:                     **else**
20:                         $V_{ACU}^{(b)} \leftarrow (1 - \alpha\lambda) V_{ACU}^{(b)} + \alpha E^{\top} U_i$.
21:                     **end if**
22:                 **end if**
23:                 $R_{ACU}^{(b)} = U_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^{\top}$.
24:                 Find orthonomal matricies $U_{ACU}^{(b)}$, $V_{ACU}^{(b)}$ and diagonal matrix $S_{ACU}^{(b)}$ such that $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^{\top} \approx R_{ACU}^{(b)}$.
25:                 Set $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$.
26:                 $j = j + 1$.
27:             **end if**
28:         **end while**
29:     **end for**
30: **end while**

Algorithm 2 Item_Vectors_Testing_Error(R, R_{ACU}, ζ)

1: Let m be the number of users (row dimension of R), n be the items (column dimension of R and R_{ACU}) and s be the number of ACUs (row dimension of R_{ACU}).
2: Initialize constants β, γ.
3: **for** $b = 0$ to ζ **do**
4: Find orthonomal $\left(\frac{s}{\zeta} \times k\right)$ matrix $U_{ACU}^{(b)}$, orthonomal $(n \times k)$ matrix $V_{ACU}^{(b)}$ and diagonal $(k \times k)$ matrix $S_{ACU}^{(b)}$ such that $U_{ACU}^{(b)} S_{ACU}^{(b)} \left(V_{ACU}^{(b)}\right)^{\top} \approx R_{ACU}^{(b)}$ and $k \leq \min\left(\frac{s}{\zeta}, n\right)$.
5: **end for**
6: Let $V_{ACU} = \left[V_{ACU}^{(0)} \cdots V_{ACU}^{(\zeta)}\right]$.
7: Aggregate $S_{ACU}^{(b)}$ for $b = 0$ to ζ along diagonal blocks to form the diagonal matrix S_{ACU}.
8: Set $V_{ACU}^{(b)} = V_{ACU}^{(b)} S_{ACU}^{(b)}$.
9: Randomly select 80% of the nonzero entries in R as training entries and let the other 20% be the probe entries.
10: Define $R^{(T)}$ as the sparse matrix made up of the training entries of R, and define $R^{(P)}$ as the sparse matrix made up of the probe entries of R such that $R = R^{(T)} + R^{(P)}$.
11: Initialize U.
12: **while** not done **do**
13: Compute sparse $E = R^{(T)} - UV_{ACU}^{\top}$ for the specified entries of $R^{(T)}$. Other entries of E remain zero.
14: **if** not done **then**
15: $U \leftarrow (1 - \beta\gamma)U + \beta E V_{ACU}$.
16: Normalize the columns of U.
17: **end if**
18: **end while**
19: Compute sparse $E = R^{(P)} - UV_{ACU}^{\top}$ for the specified entries of $R^{(P)}$.
20: Return the average absolute value of an entry in the E for the specified entries of $R^{(P)}$.

Algorithm 3 Means_Update(A, B)

1: Initialize lists L_i for $i = 0$ to the row dimension of B.
2: **for** $r = 0$ to the row dimension of A **do**
3: min_val = ∞, min_index = 0.
4: **for** $l = 0$ to the row dimension of B **do**
5: **if** The distance between the rth row of A and the lth row of B is less than min_val **then**
6: min_val = this distance.
7: min_index = l.
8: **end if**
9: **end for**
10: $L_{\text{min_index}}.push(r)$.
11: **end for**
12: **for** $l = 0$ to the row dimension of B **do**
13: **if** list L_l is non-empty **then**
14: Set the lth row of B to the weighted average of the rows of A with indices stored in list L_l.
15: **end if**
16: **end for**

Algorithm 4 Sparse_Means_Error(R, R_{ACU})

1: avg_error = 0.
2: entry_count = 0.
3: **for** $r = 0$ to the row dimension of R **do**
4: min_val = ∞, min_index = 0.
5: **for** $l = 0$ to the row dimension of R_{ACU} **do**
6: **if** The sparse distance between the rth row of R and the lth row of R_{ACU} is less than min_val for specified entries of R **then**
7: min_val = this distance.
8: min_index = l.
9: **end if**
10: **end for**
11: **for** sparse entries in the rth row of R **do**
12: entry_count += 1.
13: temp_err = absolute difference between entry of the rth row of R and the corresponding entry in the min_index row of R_{ACU}.
14: avg_error += (temp_err - avg_error) / entry_count.
15: **end for**
16: **end for**
17: Return avg_error.

4.3 Experimental Results

My main objective in my experiments is comparability across methods rather than state-of-the-art performance. To make the error on the probe entry sets comparable when testing with Algorithm 2, I aimed for similar errors on the training set; to accomplish this goal I did stop running the algorithm early when necessary.

All experiments are run on the Bridges' NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center. I ran experiments using the MovieLens ml-20m dataset (Harper and Konstan 2015). I normalized the variance and mean centered the rows of the dataset ratings matrix before proceeding.[4]

I will compare my work for generating ACUs to existing methods for selecting Core Users from (Cao and Kuang 2016; Kuang, Cao, and Chen 2017; Z. Li, Lei, and Shuyan 2016; Zeng et al. 2014). All of the methods for finding Core Users tested here are derived from the K-nearest neighbor algorithm. These methods use some metric to determine the pair-wise similarity between all existing users. Then, for each user, a list of the top-K most similar users is generated; from these combined lists the Core Users are selected. In my Core User experiments, I collect the top-50 most similar users for each user. Existing methods differ in their pair-wise similarity metric for users, and in their selection method within the compiled top-K most similar user lists.

Recall that the cosine similarity of two vectors A and B is $(A \cdot B)/(||A|| \cdot ||B||)$. In all of my Core User experiments, the pair-wise similarity between all existing users will be calculated in one of two ways: it will either be the cosine similarity of the users' ratings vectors, or it will be the cosine similarity of their boolean vectors where their boolean vectors indicate only whether or not an item has been rated - not how well the user liked it.

Once the lists of the top-K most similar users to each user have been generated, one can either count the frequency of a given user's appearance in the top-K lists and

[4]Any adjustments made here can be accounted for at recommendation time.

take the most frequent users as the Core Users, or one can weight a user's appearance in a list by the inverse of the rank within the list that user appeared; the first way I will refer to as frequency-based and the second way I will refer to as rank-based.

The final variant is whether or not to consider 'hidden' ratings in the cosine similarity of users. Without considering hidden ratings, two users with no overlapping rated items would have zero similarity, but if the users have rated items that are similar to one another this metric seems insufficient. Cao and Kuang 2016; Kuang, Cao, and Chen 2017 consider semantic relationships between items to determine item similarity, here I will use the cosine similarity of the item vectors where the item vectors are the rating data from all of the users for the given item.[5] After computing the item similarity, a missing rating may be substituted for with a weighted average of similar rated items, where the item similarity can be used as the weight.

4.3.1 MovieLens

The MovieLens ml-20m contains ratings for 138493 users on a set of 27278 movies (Harper and Konstan 2015). This section will discuss the performance of ACUs compared to existing Core User methods using the MovieLens ml-20m dataset.

Table 4.1 shows the performance of the various previously existing Core User selection methods when tested using the item vectors testing error; each method was used to collect 13000 Core Users or a little under 10% of the original MovieLens ml-20m dataset size. This is half the number of Core Users necessary to maintain recommendation accuracy as claimed in (Zeng et al. 2014), but my aim is comparing the viability of these methods. To test each method I used 13 row blocks, or 1000 users per block. I found that the performance is sensitive to the number of item vectors retained as latent factors in the line 4 of Algorithm 2; we'll discuss reasons for this after a bit further down. I chose to use 20% of the singular values, for a total

[5]For each item, a list of the top-K most similar items is generated.

Table 4.1: Item Vectors Testing Error of 13000 Core Users collected with previously existing methods using the ml-20m dataset (Harper and Konstan 2015). I averaged the results of Algorithm 2 over 75 runs where each run was given 200 independent testing users and 2600 of the Core User Item Vectors, or $\approx 50\%$ of the singular value mass. In each run, I stopped Algorithm 2 when I reached a training error of 0.2.

Item Similarity Used	Ratings Used	Frequency-Based	Probe Entry Set Error
yes	yes	yes	1.41
yes	yes	no	**1.38**
yes	no	yes	1.71
yes	no	no	1.67
no	yes	yes	1.45
no	yes	no	1.39
no	no	yes	1.73
no	no	no	1.70

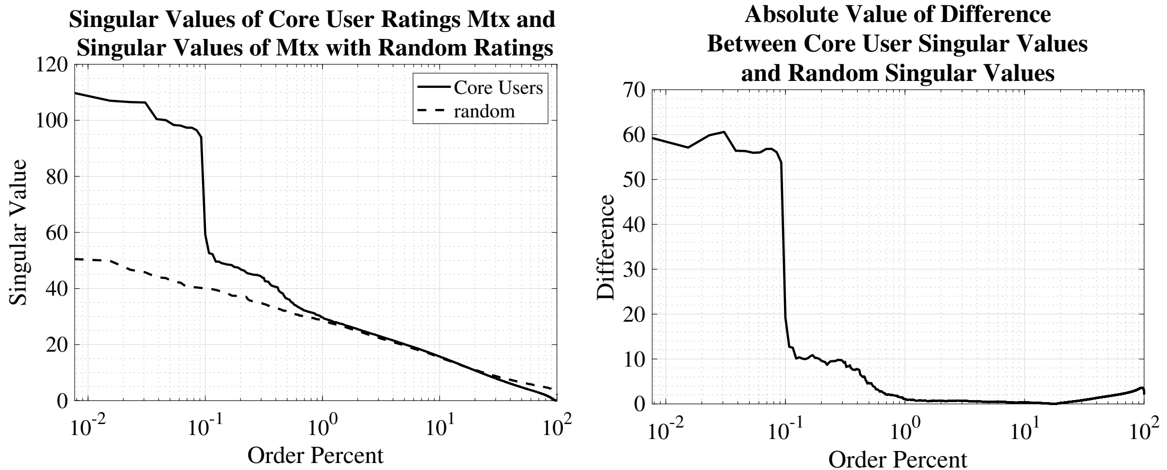

Figure 4.1: (Left) Singular Values of 13000 Core User Ratings Matrix where Core Users are selected from the ml-20m dataset (Harper and Konstan 2015) using the most competitive selection method: the method used in the second row of Table 4.1. Singular Values of an equally sparse matrix with random ratings as the non-zero values. There are 13000 singular values, where by convention lower order singular values have larger value. The x-axis is labeled as the order percent, so the ith singular value would have x-tick value $i/130$. (Right) Absolute Value of the difference between the curves in the Left Figure.

of 2600 latent factors across all the blocks. To make the error on the probe entry sets comparable across methods, I stopped running Algorithm 2 when the error on the training set had reached 0.2. The average absolute value of an entry in the ratings matrix, after centering, is ≈ 0.8; in other words, the error when always predicting that a missing rating will be the mean, zero, is ≈ 0.8. Therefore, I ran Algorithm 2 for various Core User methods until the error on the training set was $\approx 75\%$ better than simply always guessing the mean. The first column in Table 4.1 indicates whether or not hidden ratings taken from the item similarities were incorporated into the calculation of the user similarities. The second column indicates whether the cosine similarity of the user vectors is taken using the actual item ratings or just the item booleans. The third column indicates whether I used a frequency-based or a rank-based approach to select the Core Users from the aggregated lists of the top-50 most similar users to each other user. These results support previous literature suggesting that the best method for selecting Core Users considers item similarity, compares ratings rather than booleans, and uses a rank-based selection approach.

For all methods used in Table 4.1, Algorithm 4 returns an error of ≈ 0.715; this error is 12% better than the error when using only one centroid with the mean at the origin.

Figures 4.2 help to explain why the performance of Algorithm 2 is sensitive to the number of item vectors retained as latent factors in the line 4. They compare the singular values of the 13000 Core User ratings matrix where the Core Users are selected using the most competitive selection method: the method used in the second row of Table 4.1 to the singular values of an equivalently sparse matrix with random ratings as the non-zero values. In Section 2.4, I discussed the significance of singular values and how they help to explain the behavior of a matrix as a mapping between spaces. I referenced the Marchenko-Pastur theorem (Marchenko and Pastur 1967), which describes the asymptotic behavior of the singular values of large random

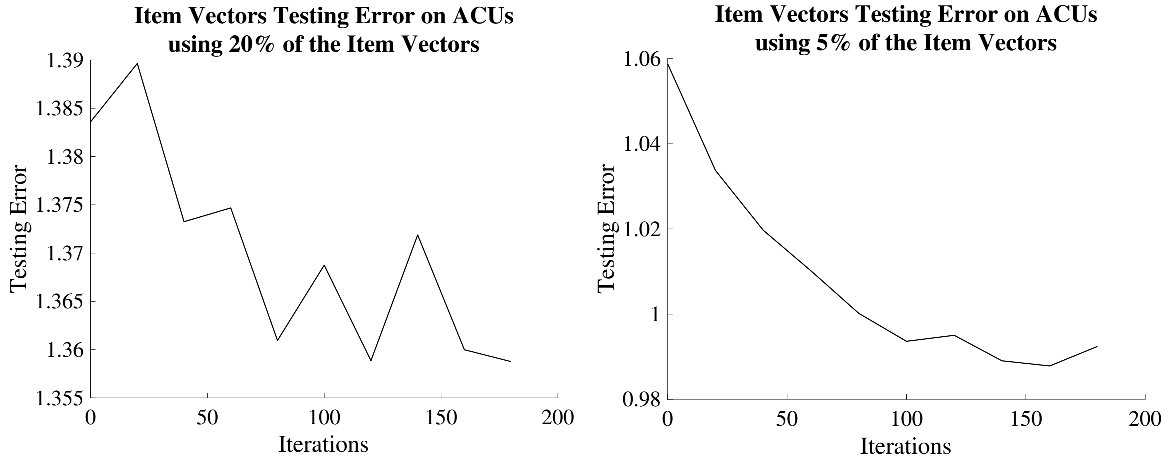

Figure 4.2: Item Vectors Testing Error of 13000 ACUs over iterations using the ml-20m dataset (Harper and Konstan 2015). For each test, I averaged the results of Algorithm 2 over 20 runs where each run was given 200 independent testing users and (Left) 2600/(Right) 650 ACU item vectors. In each run, I stopped Algorithm 2 when I reached a training error of 0.2.

matrices. Figures 4.2 show that the singular values of the Core User ratings matrix only significantly differ from those of a random matrix in the lowest order singular values, which are, by convention, the largest singular values. In fact, beyond the first 1% of singular values, the singular values of the Core User ratings matrix differ by at most 3.6 from those of a random matrix, with larger order singular values contributing less influence over the behavior of the matrix. So, while the larger order singular values of the Core User ratings matrix are non-zero, which is generally how we measure relevance, this relationship suggests that the larger order singular vectors may not be informative and may actually make learning more difficult by adding noise.

I now move to discussing the results of learning ACUs using Algorithm 1. I initialized my ACUs are as Core Users selected with the most competitive selection method: the method used in the second row of Table 4.1.

Figures 4.2 shows the item vectors testing error of 13000 ACUs over iterations, where an by an iteration of Algorithm 1 I am referring to one loop beginning on line 3. For each test, I averaged the results of Algorithm 2 over 20 runs where each run

was given 200 independent testing users and either 2600 or 650 ACU item vectors. As in my Core User tests, in each run I stopped Algorithm 2 when I reached a training error of 0.2. In both testing and training I used 13 row blocks, or 1000 ACUs per block. Whether we use 2600 or 650 ACU item vectors we can see clear improvement compared to the real Core User item vectors testing error, which is simply the $y-$intercept of these graphs. The best item vectors testing error using 650, or 5% of the 13000 ACU item vectors is better than the item vectors testing error of 27000 Core Users (20% of the users in the complete dataset) when using using 650 Core User item vectors. The best item vectors testing error using 650 of the 13000 ACU item vectors is 0.987, while the item vectors testing error of 27000 Core Users when using using 650 Core User item vectors is 1.03.

Since the Core Users are stored in the same memory format as the original complete dataset, retaining only 20% of the users as Core Users results in approximately a 80% memory reduction. The memory reduction of the Artificial Core Users depends on the number of latent factors one decides to store. If one chooses to store only 5% of the resulting latent factors, both user and item vectors, then this results in a 36% reduction in memory compared to the original data set. If one chooses to store only 5% of the the item vectors then this results in a 57% reduction in memory compared to the original data set. Additionally, the improvement in the item vectors testing error when using only 5% of the the item vectors in Algorithm 2 compared to using 20% of the the item vectors as shown in Figures 4.2 suggests that these extra vector may be more noisy than informative.

Admittedly, Algorithm 1 learns relatively slowly. One loop beginning on line 3 can take up to 4 minutes to complete. I stopped running Algorithm 1 after 180 iterations at which point I reached an item vectors testing error of 1.36 with 2600 ACU item vectors and 0.99 with 650 ACU item vectors, which outperforms the most competitive real Core User methods. I also improved the sparse mean error *slightly*,

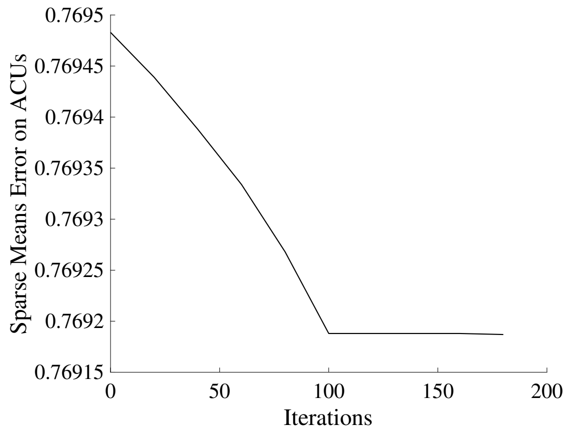

Figure 4.3: Sparse Mean Error of 13000 ACUs using the ml-20m dataset (Harper and Konstan 2015).

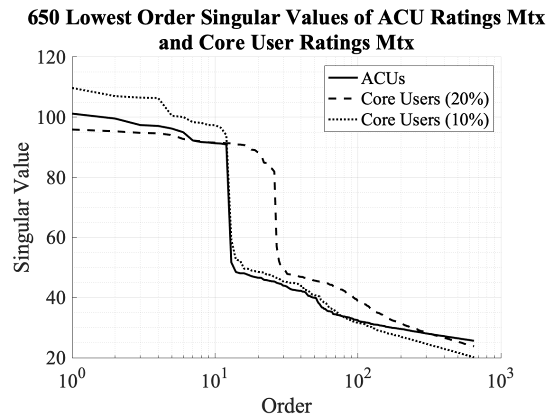

Figure 4.4: 650 largest singular values of 13000 trained ACU ratings matrix using the ml-20m dataset (Harper and Konstan 2015), compared to the largest singular values of the ratings matrix of 27000 Core Users (20% of the users in the complete dataset) and the largest singular values of the ratings matrix of 13000 Core Users (10% of the users in the complete dataset).

so my ACUs are a bit better centroids for the testing users than 13000 real Core Users are. Figure 4.3 shows the sparse mean error of my ACUs over iterations calculated with Algorithm 4. The second improvement is marginal, but demonstrates that my ACUs are still resemble an average group real users.

I was able to condense the information of 27000 sparse Core Users into 13000 ACUs or approximately half the number of users. Since ACUs carry dense information, storing 5% of the 13000 ACUs latent factors takes up about three times as much memory as 27000 sparse Core Users do, but 5% of the 13000 ACUs latent factors achieves a smaller item vectors testing error than the same number of Core User latent vectors when using 27000 sparse Core Users. We can see from Figure 4.4 that

the largest singular values of the ACU ratings matrix, that had matched the largest singular values of the ratings matrix with 13000 Core Users at initialization, has shifted toward the the largest singular values of the ratings matrix with 27000 Core Users after 180 training iterations.

4.4 Discussion

I have shown that my Artificial Core Users improve the recommendation accuracy of real Core Users while mimicking real user data. Because they act as good centroids for the complete dataset, they can be considered good representatives for real user clusters. They can be stored efficiently, yet they have dense ratings information which is more immediately interpretable than sparse data. I have removed the representation limits of Core Users and shown that an iterative training process can improve the recommendation accuracy of Core Users producing data that continues to resemble that of real users, conserve memory and improve recommendation efficiency.

CHAPTER V

Conclusion

The immense expressional power of machine learning and has advanced data sciences a great deal. Large networks can achieve unprecedented accuracy in intricate learning problems, yet their size consumes significant computational resources and, consequently, time (Krizhevsky, Sutskever, and G. E. Hinton 2012). Advances in compute power allow neural networks with millions of parameters to be trained on enormous, complex data sets, and the use of GPUs has decreased training time drastically, but new techniques for reducing network training time must arise for deep learning to progress. Ensemble recommender systems can make sense of human preferences for datasets with millions of users and items (*The Netflix Prize* Retrieved 2009-9-24), but the vast majority of this work is non-transferable, which is a significant obstruction.

In this thesis, I have introduced three methods to improve the efficiency of machine learning methods with special focus given to computationally heavy algorithms such as deep learning and recommender systems. I have shown that Block Diagonal Inner Product layers can reduce network size, training time and final execution time without significant harm to the network performance. I have developed a general adaptation to gradient descent that reduces the number of iterations required to learn by incorporating traditional predictor-corrector inspired ideas into classic

gradient descent. And I have shown that Artificial Core Users improve the recommendation performance of real Core Users while remaining good representation of an average collection of real users. Together, these works contribute to a growing need to make behemoth learning models more efficient to train, store and execute.

APPENDICES

APPENDIX A

Lenet-5

Below you'll find the Lenet-5 (LeCun, Bottou, et al. 1998) network architecture written explicitly with all activation functions and initialization distributions specified. Caffe (Jia et al. 2014) reads Google Protobuf (*Language Guide* Retrieved 2020-07-31) format files, so below I have written the Lenet-5 network architecture in this format. The Lenet-5 network has 5 layers by conventional standards including the input layer, however Caffe reads in pooling and activation functions as separate layers so I adhere to Caffe's format below. Additionally, the input and output layers have two separate protobufs, one for training and one for testing, that specify different actions for each phase, like batch sizes for example.

This protobuf is identical to the one used in *Training LeNet on MNIST with Caffe* Retrieved 2020-07-31, and a more detailed explanation can be found there. The input layer pixels are scales so that they are in the range $[0, 1)$. The 'bottom' field specifies the name of the input for that layer, and the 'top' field gives a name to the layer output. Of note are parameter values for the num_output, filler, kernel_size and stride fields. The xavier filler algorithm automatically determines the scale of initialization based on the number of input and output neurons (Glorot and Y. Bengio 2010). When the filler is 'constant', the default value is 0. lr_mults values adjustment the learning rate for the layer's learnable parameters. In a convolutional layer, the

num_output field indicated the number of output channels.

```
name: “LeNet”
layer {
    name: “dataset_name”
    type: “Data”
    top: “data”
    top: “label”
    include {phase: TRAIN}
    # 0.00390625 = 1/256
    transform_param {scale: 0.00390625 }
    data_param {
        source: “source_path/training_dataset_lmdb”
        batch_size: 64
        backend: LMDB
    }
}
layer {
    name: “dataset_name”
    type: “Data”
    top: “data”
    top: “label”
    include {phase: TEST }
    # 0.00390625 = 1/256
    transform_param {scale: 0.00390625}
    data_param {
        source: “source_path/testing_dataset_lmdb”
        batch_size: 100
        backend: LMDB
    }
}
layer {
    name: “conv1”
    type: “Convolution”
    bottom: “data”
    top: “conv1”
    param {lr_mult: 1 }
    param {lr_mult: 2 }
    convolution_param {
        num_output: 20
        kernel_size: 5
```

```
        stride: 1
        weight_filler { type: "xavier" }
        bias_filler { type: "constant" }
    }
}
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
layer {
    name: "conv2"
    type: "Convolution"
    bottom: "pool1"
    top: "conv2"
    param {lr_mult: 1 }
    param {lr_mult: 2 }
    convolution_param {
        num_output: 50
        kernel_size: 5
        stride: 1
        weight_filler { type: "xavier" }
        bias_filler { type: "constant"}
    }
}
layer {
    name: "pool2"
    type: "Pooling"
    bottom: "conv2"
    top: "pool2"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
```

```
}
layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "pool2"
    top: "ip1"
    param {lr_mult: 1 }
    param {lr_mult: 2}
    inner_product_param {
        num_output: 500
        weight_filler { type: "xavier" }
        bias_filler { type: "constant"}
    }
}
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"
}
layer {
    name: "ip2"
    type: "InnerProduct"
    bottom: "ip1"
    top: "ip2"
    param {lr_mult: 1 }
    param {lr_mult: 2}
    inner_product_param {
        num_output: 10
        weight_filler { type: "xavier" }
        bias_filler { type: "constant"}
    }
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {phase: TEST }
}
layer {
```

```
        name: "loss"
        type: "SoftmaxWithLoss"
        bottom: "ip2"
        bottom: "label"
        top: "loss"
}
```

Below you'll find a copy of the corresponding 'solver' protobuf. By Caffe's design (Jia et al. 2014), the solver contains hyperparameter values like learning rate, testing frequency, how often to save a network snapshot and learning policies like momentum and weight decay.

```
# The train/test net protocol buffer definition
net: "source_path/net_protocol_buffer.prototxt"
# test_iter specifies how many forward passes the test should carry out.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
# momentum is the weight of the previous update.
momentum: 0.9
# weight_decay meta parameter governs the regularization term.
weight_decay: 0.005
# The learning rate policy
# inv: return base_lr * (1 + gamma * iter) ^(- power)
lr_policy: "inv" gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations (at this learning rate)
# After which reduce the learning rate by a factor of 10
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "save_path/snapshot_prefix"
# solver mode: CPU or GPU
solver_mode: GPU
```

APPENDIX B

Cuda-Convnet

Below you'll find Krizhevsky's Cuda-convnet (Krizhevsky 2012a) network architecture written explicitly with all activation functions and initialization distributions specified. Caffe (Jia et al. 2014) reads Google Protobuf (*Language Guide* Retrieved 2020-07-31) format files, so below I have written Krizhevsky's Cuda-convnet network architecture in this format. Krizhevsky's Cuda-convnet network has 6 layers by conventional standards including the input layer, however Caffe reads in pooling and activation functions as separate layers so I adhere to Caffe's format below. Additionally, the input and output layers have two separate protobufs, one for training and one for testing, that specify different actions for each phase, like batch sizes for example.

This protobuf is identical to the one used in *Alex's CIFAR-10 tutorial, Caffe style* Retrieved 2020-07-31, and a more detailed explanation can be found there. The 'bottom' field specifies the name of the input for that layer, and the 'top' field gives a name to the layer output. Of note are parameter values for the num_output, filler, kernel_size, pad and stride fields. When the filler is 'constant', the default value is 0. lr_mults values adjustment the learning rate for the layer's learnable parameters. In a convolutional layer, the num_output field indicated the number of output channels.

```
name: “Cuda-convnet”
layer {
    name: “dataset_name”
    type: “Data”
    top: “data”
    top: “label”
    include {phase: TRAIN}
    transform_param {mean_file: “source_path/mean.binaryproto” }
    data_param {
        source: “source_path/training_dataset_lmdb”
        batch_size: 100
        backend: LMDB
    }
}
layer {
    name: “dataset_name”
    type: “Data”
    top: “data”
    top: “label”
    include {phase: TEST }
    transform_param {mean_file: “source_path/mean.binaryproto” }
    data_param {
        source: “source_path/testing_dataset_lmdb”
        batch_size: 100
        backend: LMDB
    }
}
layer {
    name: “conv1”
    type: “Convolution”
    bottom: “data”
    top: “conv1”
    param {lr_mult: 1 }
    param {lr_mult: 2 }
    convolution_param {
        num_output: 32
        pad: 2
        kernel_size: 5
        stride: 1
        weight_filler {
            type: “gaussian”
            std: 0.0001
```

```
        }
        bias_filler { type: "constant" }
    }
}
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "pool1"
    top: "pool1"
}
layer {
    name: "conv2"
    type: "Convolution"
    bottom: "pool1"
    top: "conv2"
    param {lr_mult: 1 }
    param {lr_mult: 2 }
    convolution_param {
        num_output: 32
        pad: 2
        kernel_size: 5
        stride: 1
        weight_filler {
            type: "gaussian"
            std: 0.01
        }
        bias_filler { type: "constant"}
    }
}
layer {
```

```
        name: “relu2”
        type: “ReLU”
        bottom: “conv2”
        top: “conv2”
}
layer {
        name: “pool2”
        type: “Pooling”
        bottom: “conv2”
        top: “pool2”
        pooling_param {
                pool: AVE
                kernel_size: 3
                stride: 2
        }
}
layer {
        name: “conv3”
        type: “Convolution”
        bottom: “pool1”
        top: “conv3”
        param {lr_mult: 1 }
        param {lr_mult: 2 }
        convolution_param {
                num_output: 64
                pad: 2
                kernel_size: 5
                stride: 1
                weight_filler {
                        type: “gaussian”
                        std: 0.01
                }
                bias_filler { type: “constant”}
        }
}
layer {
        name: “relu3”
        type: “ReLU”
        bottom: “conv3”
        top: “conv3”
}
layer {
```

```
        name: “pool3”
        type: “Pooling”
        bottom: “conv3”
        top: “pool3”
        pooling_param {
                pool: AVE
                kernel_size: 3
                stride: 2
        }
}
layer {
        name: “ip1”
        type: “InnerProduct”
        bottom: “pool3”
        top: “ip1”
        param {lr_mult: 1 }
        param {lr_mult: 2}
        inner_product_param {
                num_output: 64
                weight_filler {
                        type: “gaussian”
                        std: 0.1
                }
                bias_filler { type: “constant”}
        }
}
layer {
        name: “ip2”
        type: “InnerProduct”
        bottom: “ip1”
        top: “ip2”
        param {lr_mult: 1 }
        param {lr_mult: 2}
        inner_product_param {
                num_output: 10
                weight_filler {
                        type: “gaussian”
                        std: 0.1
                }
                bias_filler { type: “constant”}
        }
```

```
}
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {phase: TEST }
}
layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}
```

Below you'll find a copy of the corresponding 'solver' protobuf. By Caffe's design (Jia et al. 2014), the solver contains hyperparameter values like learning rate, testing frequency, how often to save a network snapshot and learning policies like momentum and weight decay.

```
# reduce the learning rate after 8 epochs (4000 iters) by a factor of 10

# The train/test net protocol buffer definition
net: "source_path/net_protocol_buffer.prototxt"
# test_iter specifies how many forward passes the test should carry out.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.001
# momentum is the weight of the previous update.
momentum: 0.9
# weight_decay meta parameter governs the regularization term.
weight_decay: 0.004
# The learning rate policy
lr_policy: "fixed"
# Display every 100 iterations
display: 100
```

```
# The maximum number of iterations (at this learning rate)
# After which reduce the learning rate by a factor of 10
max_iter: 4000
# snapshot intermediate results
snapshot: 4000
snapshot_prefix: “save_path/snapshot_prefix”
# solver mode: CPU or GPU
solver_mode: GPU
```

APPENDIX C

Proof of Theorem 3.3.1 and Corollary 3.3.1.1

Recall, L is *β-smooth* if the gradient mapping ∇L is β-Lipschitz, i.e. for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ one has $\|\nabla L(\boldsymbol{x}) - \nabla L(\boldsymbol{y})\| \leq \beta \|\boldsymbol{x} - \boldsymbol{y}\|$. Recall the follow two known lemmas.

Lemma C.0.1. *Let L be convex and β-smooth on $\mathbb{R}^n$, then for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ we have $|L(\boldsymbol{x}) - L(\boldsymbol{y}) - \nabla L(\boldsymbol{y})^\top (\boldsymbol{x} - \boldsymbol{y})| \leq \beta \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$.*

Proof. Using the convexity of L, the Cauchy-Schwarz inequality and the fact that L is β-smooth we have,

$$0 \leq L(\boldsymbol{x}) - L(\boldsymbol{y}) - \nabla L(\boldsymbol{y})^\top (\boldsymbol{x} - \boldsymbol{y}) \leq \left(\nabla L(\boldsymbol{x})^\top - \nabla L(\boldsymbol{y})^\top\right)(\boldsymbol{x} - \boldsymbol{y}) \leq \beta \|\boldsymbol{x} - \boldsymbol{y}\|_2^2 .$$

$\square$

Lemma C.0.2. *Let L be convex and β-smooth on $\mathbb{R}^n$, then for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ we have $L\left(\boldsymbol{x} - \frac{1}{\beta}\nabla L(\boldsymbol{x})\right) - L(\boldsymbol{y}) \leq -\frac{1}{2\beta}\|\nabla L(\boldsymbol{x})\|_2^2 + \nabla L(\boldsymbol{x})^\top (\boldsymbol{x} - \boldsymbol{y})$.*

Proof. Using the convexity of L and Lemma C.0.1 we have,

$$\begin{aligned} L\left(\boldsymbol{x}-\frac{1}{\beta}\nabla L(\boldsymbol{x})\right)-L(\boldsymbol{y}) \leq & L\left(\boldsymbol{x}-\frac{1}{\beta}\nabla L(\boldsymbol{x})\right)-L(\boldsymbol{x})+\nabla L(\boldsymbol{x})^\top(\boldsymbol{x}-\boldsymbol{y}) \\ \leq & \nabla L(\boldsymbol{x})^\top\left(\boldsymbol{x}-\frac{1}{\beta}\nabla L(\boldsymbol{x})-\boldsymbol{x}\right) \\ & +\frac{\beta}{2}\left\|\boldsymbol{x}-\frac{1}{\beta}\nabla L(\boldsymbol{x})-\boldsymbol{x}\right\|_2^2+\nabla L(\boldsymbol{x})^\top(\boldsymbol{x}-\boldsymbol{y}) \\ = & -\frac{1}{2\beta}\|\nabla L(\boldsymbol{x})\|_2^2+\nabla L(\boldsymbol{x})^\top(\boldsymbol{x}-\boldsymbol{y})\ . \end{aligned}$$

□

I will now prove Theorem 3.3.1 and Corollary 3.3.1.1.

Proof of Theorem 3.3.1. From Lemma C.0.2 we have,

$$\begin{aligned} L\left(\boldsymbol{\theta}_{t+1}\right)-L(\boldsymbol{\theta}_{t-r+1}) = & L\left(\boldsymbol{z}_t-\frac{1}{\beta}\nabla L(\boldsymbol{z}_t)\right)-L(\boldsymbol{\theta}_{t-r+1}) \\ \leq & -\frac{1}{2\beta}\|\nabla L(\boldsymbol{z}_t)\|_2^2+\nabla L(\boldsymbol{z}_t)^\top(\boldsymbol{z}_t-\boldsymbol{\theta}_{t-r+1}) \qquad \text{(C.1)} \\ = & -\frac{\beta}{2}\|\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t\|_2^2-\beta(\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t)^\top(\boldsymbol{z}_t-\boldsymbol{\theta}_{t-r+1})\ . \end{aligned}$$

Similarly,

$$L\left(\boldsymbol{\theta}_{t+1}\right)-L(\boldsymbol{\theta}^*) \leq -\frac{\beta}{2}\|\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t\|_2^2-\beta(\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t)^\top(\boldsymbol{z}_t-\boldsymbol{\theta}^*)\ . \qquad \text{(C.2)}$$

Multiplying (C.1) by $(\lambda_t - 1)$ and adding that to (C.2) we get

$$\begin{aligned} \lambda_t\delta_{t+1}-(\lambda_t-1)\delta_{t-r+1} \leq & -\frac{\beta\lambda_t}{2}\|\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t\|_2^2 \\ & -\beta(\boldsymbol{\theta}_{t+1}-\boldsymbol{z}_t)^\top(\lambda_t\boldsymbol{z}_t-(\lambda_t-1)\boldsymbol{\theta}_{t-r+1}-\boldsymbol{\theta}^*) \end{aligned}$$

where $\delta_t = L(\boldsymbol{\theta}_t) - L(\boldsymbol{\theta}^*)$. Now I multiply both sides by λ_t and note that by definition

$\lambda_t^2 - \lambda_t = \lambda_{t-r}^2$. This gives,

$$\begin{aligned}\lambda_t^2\delta_{t+1} - \lambda_{t-r}^2\delta_{t-r+1} \leq & -\frac{\beta}{2}\Big(\|\lambda_t(\boldsymbol{\theta}_{t+1} - \boldsymbol{z}_t)\|_2^2 \\ & + 2\lambda_t(\boldsymbol{\theta}_{t+1} - \boldsymbol{z}_t)^\top(\lambda_t\boldsymbol{z}_t - (\lambda_t - 1)\boldsymbol{\theta}_{t-r+1} - \boldsymbol{\theta}^*)\Big) \\ = & -\frac{\beta}{2}\Big(\|\lambda_t\boldsymbol{\theta}_{t+1} - (\lambda_t - 1)\boldsymbol{\theta}_{t-r+1} - \boldsymbol{\theta}^*\|_2^2 \\ & + \|\lambda_t\boldsymbol{z}_t - (\lambda_t - 1)\boldsymbol{\theta}_{t-r+1} - \boldsymbol{\theta}^*\|_2^2\Big)\ .\end{aligned} \tag{C.3}$$

Rearranging the update for the $\boldsymbol{z}_t$ I note, $\lambda_{t+1}\boldsymbol{z}_{t+1} - (\lambda_{t+1} - 1)\boldsymbol{\theta}_{t-r+2} = \lambda_t\boldsymbol{\theta}_{t+1} - (\lambda_t - 1)\boldsymbol{\theta}_{t-r+1}$. Substituting this into (C.3) yeilds,

$$\begin{aligned}\lambda_t^2\delta_{t+1} - \lambda_{t-r}^2\delta_{t-r+1} \leq & -\frac{\beta}{2}\Big(\|\lambda_{t+1}\boldsymbol{z}_{t+1} - (\lambda_{t+1} - 1)\boldsymbol{\theta}_{t-r+2} - \boldsymbol{\theta}^*\|_2^2 \\ & + \|\lambda_t\boldsymbol{z}_t - (\lambda_t - 1)\boldsymbol{\theta}_{t-r+1} - \boldsymbol{\theta}^*\|_2^2\Big)\ .\end{aligned}$$

By summing both sides of the above inequality from $t = r$ to T for some $T > r$ we have,

$$\sum_{t=T-r}^{T} \lambda_t^2\delta_{t+1} \leq \frac{\beta\|\boldsymbol{z}_r - \boldsymbol{\theta}^*\|_2^2}{2}$$

where it is important to note that $\lambda_{t-1} \geq \lfloor t/r \rfloor/2$ by induction. Hence,

$$\sum_{t=T-r}^{T} \lfloor (t+1)/r \rfloor^2\delta_{t+1} \leq 2\beta\|\boldsymbol{z}_r - \boldsymbol{\theta}^*\|_2^2$$

which concludes the proof. □

Proof of Corollary 3.3.1.1. Recall L is strongly convex with parameter $m > 0$ if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ we have $\frac{m}{2}\|\boldsymbol{x} - \boldsymbol{y}\|_2^2 \leq L(\boldsymbol{x}) - L(\boldsymbol{y}) - \nabla L(\boldsymbol{y})^\top(\boldsymbol{x} - \boldsymbol{y})$. With $\boldsymbol{x} = \boldsymbol{z}_r$ and $\boldsymbol{y} = \boldsymbol{\theta}^*$ this means,

$$\|\boldsymbol{z}_r - \boldsymbol{\theta}^*\|_2^2 \leq \frac{2(L(\boldsymbol{z}_r) - L(\boldsymbol{\theta}^*))}{m} \leq \frac{\beta\|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2^2}{mr}$$

by convergence of gradient descent (Nocedal and Wright 2006) and since $\nabla L(\boldsymbol{\theta}^*) = 0$. Together with Theorem 3.3.1 this gives,

$$\sum_{t=T-r}^{T} \lfloor (t+1)/r \rfloor^2 \delta_{t+1} \leq \frac{\beta^2\|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2^2}{mr}$$

as desired. □

APPENDIX D

NAG-PCGD

NAG-PCGD nests Nesterov's Accelerated Gradient (NAG) updates inside Predictor-Corrector Gradient Descent (PCGD) updates to achieve the benefits of both methods. Let p be the prediction increment, s be the snapshot increment and let $f(\boldsymbol{a}, x) : \mathbb{R}^c \times \mathbb{R} \to \mathbb{R}$ be our chosen fit function class for parameter prediction. Recall, $F(A, x) : \mathbb{R}^{c \times n} \times \mathbb{R} \to \mathbb{R}^n$ is defined such that the ith entry of $F(A, x)$ is $f(\boldsymbol{a}_i, x)$ where $\boldsymbol{a}_i$ is the ith column of A. When using NAG-PCGD, network parameter vector $\boldsymbol{\theta} \in \mathbb{R}^n$ receives the update

$$\begin{aligned}
\lambda_t &= \left(1 + \sqrt{1 + 4\lambda_{t-1}^2}\right) \Big/ 2 \qquad \text{with } \lambda_0 = 0 \\
\gamma_t &= (1 - \lambda_t) / \lambda_{t+1} \\
\boldsymbol{z}_t &= (1 - \gamma_{t-1})\boldsymbol{\theta}_t + \gamma_{t-1}\boldsymbol{\theta}_{t-1} \qquad \text{with } \boldsymbol{z}_0 = \boldsymbol{\theta}_0 \\
\boldsymbol{\theta}_{t+1} &= \begin{cases} F(A_{t+1}, l_{t+1}) & \text{if } \ t+1 \equiv 0 \mod p \\ \boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t) & \text{otherwise} \end{cases}
\end{aligned} \tag{D.1}$$

where L is the desired loss function, ϵ is some learning rate, $l_{t+1} \geq p/s$ is an increasing prediction length and $A_{t+1} \in \mathbb{R}^{c \times n}$, minimizes the L_2-norms of the columns of $JA_{t+1} - \Theta_{t+1}$. Here, $J_{i,j} = \partial f(\boldsymbol{a}, i) \big/ \partial a_j$, and the ith row of Θ_{t+1} is the vector $\boldsymbol{\theta}_{t+1-p+is}^{\top}$ for

$i < p/s$ and $(\boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t))^\top$ for $i = p/s$.

BIBLIOGRAPHY

BIBLIOGRAPHY

AGA News: Kim Prevails Again In Man Vs Machine Rematch (Retrieved 2009-08-08). URL: `https://www.usgo.org/news/`.

Aggarwal, Charu C. (2016). *Recommender Systems*. Springer International Publishing Switzerland.

Ailon, N. and B. Chazelle (2009). "The Fast Johnson Lindenstrauss Transform and approximate nearest neighbors". In: *IAM Journal on Computing* 39.1, pp. 302–322.

Alex's CIFAR-10 tutorial, Caffe style (Retrieved 2020-07-31). URL: `https://caffe.berkeleyvision.org/gathered/examples/cifar10.html`.

Andrychowicz, M. et al. (2016). "Learning to learn by gradient descent by gradient descent". In: *NIPS*.

Beck, A. and M. Teboulle (2009). "A fast iterative shrinkage-thresholding algorithm for linear inverse problems". In: *Siam Journal Imaging Sciences* 2.1, pp. 183–202.

Boahen, K. (2014). "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations". In: *IEEE* 102.5, pp. 699–716.

Breiman, Leo (1998). "Arcing classifier (with discussion and a rejoinder by the author)". In: *Ann. Stat.* 26, pp. 801–849.

Cao, Gaofeng and Li Kuang (2016). "Identifying Core Users based on Trust Relationships and Interest Similarity in Recommender System". In: *IEEE International Conference on Web Services*.

Cassioli, A. et al. (2013). "An incremental least squares algorithm for large scale linear classification". In: *European Journal of Operational Research* 224.3, pp. 560–565.

Cheng, Y. et al. (2015). "An exploration of parameter redundancy in deep networks with circulant projections". In: *ICCV*.

Chollet, François (2017). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *arXiv*. URL: `arXiv:1610.02357`.

Christakopoulou, Konstantina and Arindam Banerjee (2018). "Adversarial Recommendation: Attack of the Learned Fake Users". In: *arXiv*. URL: `arXiv:1809.08336`.

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2015). "Low-precision Storage for Deep Learning". In: *ICLR*.

Daniel, C., J. Taylor, and S. Nowozin (2016). "Learning step size controllers for robust neural network training". In: *AAAI*.

Delcker, Janosch (2018). "The man who invented the self-driving car (in 1986)". In: *Politico*.

Dozat, T. (2016). "Incorporating Nesterov Momentum into Adam". In: *ICLR Workshop*.

Duchi, J., E. Hazan, and Y. Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *JMLR*.

Glorot, X. and Y. Bengio (2010). "Understanding the Difficulty of Training Deep Feedforward Neural Networks". In: *International Conference on Artificial Intelligence and Statistics* 9, pp. 249–256.

Gupta, S. et al. (2015). "Deep Learning with Limited Numerical Precision". In: *ICML*, pp. 1737–1746.

Han, S., H. Mao, and W. J. Dally (2015). "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *ICLR*.

Han, S., J. Pool, et al. (2015). "Learning both Weights and Connections for Efficient Neural Networks". In: *NIPS*, pp. 1135–1143.

Harper, F. Maxwell and Joseph A. Konstan (2015). "The MovieLens Datasets: History and Context". In: *ACM Transactions on Interactive Intelligent Systems (TiiS) 5* 4.

He, Kaiming et al. (2015). "Deep residual learning for image recognition". In: *arXiv*. URL: `arXiv:1512.03385`.

He, T. et al. (2014). "Reshaping Deep Neural Network for Fast Decoding By Node-pruning". In: *IEEE ICASSP*, pp. 245–249.

He, Xiangnan et al. (2017). "Neural collaborative filtering". In: *WWW*.

Heeger, D. J. (2016). "Theory of cortical function". In: *Proceedings of the National Academy of Sciences of the United States of America* 114.8, pp. 1773–1782.

Herculano-Houzel, S. (2012). “The Remarkable, Yet Not Extraordinary, Human Brain as a Scaled-Up Primate Brain and Its Associated Cost”. In: *NAS*.

Hinton, G., O. Vinyals, and J. Dean (2014). “Distilling the Knowledge in a Neural Network”. In: *NIPS*.

Ho, Q. et al. (2013). “More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server”. In: *NIPS*, pp. 1223–1231.

Hratchian, H., M. J. Frisch, and H. B. Schlegel (2010). “Steepest Descent Reaction Path Integration Using a First-Order Predictor-Corrector Method”. In: *The Journal of Chemical Physics* 133.22.

Hu, Y., Y. Koren, and C. Volinsky (2008). “Collaborative filtering for implicit feedback datasets”. In: *ICDM*.

Huang, Gao et al. (2016). “Densely Connected Convolutional Networks”. In: *arXiv*. URL: `arXiv:1608.06993v3`.

Ioannou, Y. et al. (2017). “Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups”. In: *CVPR*.

Jain, Prateek and Inderjit S. Dhillon (2013). “Provable Inductive Matrix Completion”. In: *arXiv*. URL: `arXiv:1306.0626`.

Jhurani, C. and P. Mullowney (2015). “A GEMM Interface and Implementation on NVIDIA GPUs for Multiple Small Matrices”. In: *Journal of Parallel and Distributed Computing* 75, pp. 133–140.

Jia, Y. et al. (2014). “Caffe: Convolutional architecture for fast feature embedding”. In: *arXiv*. URL: `arXiv:1408.5093`.

Kingma, D. and J. Ba (2015). “Adam: A Method for Stochastic Optimization”. In: *ICLR*.

Kononenko, Igor (2001). “Machine learning for medical diagnosis: history, state of the art and perspective”. In: *Artificial Intelligence in Medicine* 23, pp. 89–109.

Koren, Y. (2008). “Factorization meets the neighborhood: a multifaceted collaborative filtering model”. In: *KDD*.

Koren, Y., R. Bell, and C. Volinsky (2009). “Matrix factorization techniques for recommender systems”. In: *Computer*.

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. Computer Science, University of Toronto.

— (2012a). *cuda-convnet*. Tech. rep. Computer Science, University of Toronto.

Krizhevsky, A. (2012b). *cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks*. Tech. rep. Computer Science, University of Toronto.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NIPS*, pp. 1106–1114.

Kuang, Li, Gaofeng Cao, and Liang Chen (2017). “Extracting Core Users Based on Features of Users and Their Relationships in Recommender Systems”. In: *International Journal of Web Services Research* 14.

L2, Q., T. Sarlo, and A. Smola (2013). “Fastfood – approximating kernel expansions in loglinear time”. In: *ICML*.

Lam, Shyong K and John Riedl (2004). “Shilling recommender systems for fun and profit”. In: *WWW*, pp. 393–402.

Language Guide (Retrieved 2020-07-31). URL: `https://developers.google.com/protocol-buffers/docs/overview`.

Lebedev, V. and V. Lempitsky (2016). “Fast convnets using group-wise brain damage”. In: *CVPR*.

LeCun, Y., L. Bottou, et al. (1998). “Gradient-based Learning Applied to Document Recognition”. In: *IEEE* 86.11, pp. 2278–2324.

LeCun, Y., C. Cortes, and C. J.C. Burges (n.d.). *The MNIST Database of handwritten digits*.

Li, Qibing, Xiaolin Zheng, and Xinyue Wu (2018). “Neural Collaborative Autoencoder”. In: *arXiv*. URL: `arXiv:1712.09043`.

Li, S., J. Kawale, and Y. Fu (2015). “Deep collaborative filtering via marginalized denoising auto-encoder”. In: *CIKM*.

Li, Zhang, Yu Lei, and Cao Shuyan (2016). “Constructing the Core User Set for Collaborative Recommendation Based on Samples Selection Idea”. In: *International Journal of u- and e- Service, Science and Technology* 9, pp. 27–34.

Luca, M. D. and D. Rhodes (2016). “Optimal Perceived Timing: Integrating Sensory Information with Dynamically Updated Expectations”. In: *Scientific Reports* 6.28563.

Marchenko, V. A. and L. Pastur (1967). “Distribution of eigenvalues for some sets of random matrices”. In: *Mathematics of the USSR-Sbornik* 1.4, pp. 457–483.

Masliah, I. et al. (2016). “High-Performance Matrix-Matrix Multiplications of Very Small Matrices”. In: *Euro-Par 2016: Parallel Processing* 9833, pp. 659–671.

McDermott, Nick (2019). "Computers can be 'better than doctors at spotting disease' study finds". In: *The Sun*. URL: `https://www.thesun.co.uk/news/9999433/computers-better-doctors-diagnosis/`.

Merolla, P. A. et al. (2014). "A Million Spiking-Neuron Integrated Circuit With a Scalable Communication Network and Interface". In: *Science* 345.6197, pp. 668–673.

Moczulski, M. et al. (2016). "ACDC: A Structured Efficient Linear Layer". In: *arXiv*. URL: `arXiv:1511.05946`.

N.Tishby (June 2017). *Information Theory of Deep Learning*. Deep Learning: Theory, Algorithms, and Applications.

Neelakantan, A. et al. (2015). "Adding Gradient Noise Improves Learning for Very Deep Networks". In: *arXiv*. URL: `arXiv:1511.06807`.

Nesterov, Y. (1983). "A method of solving a convex programming problem with convergence rate O(1/sqr(k))". In: *Soviet Mathematics Doklady* 27, pp. 372–376.

Netzer, Y. et al. (2011). "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS*.

Nickolls, J. et al. (1998). "Scalable Parallel Programming with CUDA". In: *ACM Queue* 6.2, pp. 40–53.

Nocedal, J. and S. J. Wright (2006). "Numerical optimization". In: *Springer Series in Operations Research and Financial Engineering*.

OMahony, Michael P, Neil J Hurley, and Guenole CM Silvestre (2002). "Promoting recommendations: An attack on collaborative filtering". In: *International Conference on Database and Expert Systems Applications*, pp. 494–503.

Polyak, B.T. (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17.

Rajan, K. (2010). "What Do Random Matrices Tell Us About the Brain?" In: *Grace Hopper Celebration of Women in Computing*.

Rajan, K. and L. F. Abbott (2006). "Eigenvalue Spectra of Random Matrices for Neural Networks". In: *Physical Review Letters* 97.18.

Rashid, Al Mamunur, Istvan Albert, et al. (2002). "Getting to Know You: Learning New User Preferences in Recommender Systems". In: *IUI*.

Rashid, Al Mamunur, George Karypis, and John Riedl (2008). "Learning Preferences of New Users in Recommender Systems: An Information Theoretic Approach". In: *KDD* 10, pp. 90–100.

Reed, R. (1993). “Pruning Algorithms-A Survey”. In: *IEEE Transactions on Neural Networks* 4.5, pp. 740–747.

Sainath, T. N. et al. (2013). “Low-Rank Matrix Factorization for Deep Neural Network Training with High-Dimensional Output Targets”. In: *IEEE ICASSP*.

Sarwar, Badrul M. et al. (1998). “Using Filtering Agents to Improve Prediction Quality in the GroupLens Research Collaborative Filtering System”. In: *Computer Supported Cooperative Work*, pp. 345–354.

Sarwar†, Badrul et al. (2002). “Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems”. In: *ICCIT*.

Saxe, A. M. et al. (2011). “On Random Weights and Unsupervised Feature Learning”. In: *ICML*.

Schapire, Robert E. (1990). “The Strength of Weak Learnability”. In: *Machine Learning* 5, pp. 197–227.

Scieur, D., A. d’Aspremont, and F. Bach (2016). “Regularized Nonlinear Acceleration”. In: *NIPS*.

Sculley, D. (2010). “Web-Scale K-Means Clustering”. In: *WWW*, pp. 1177–1178.

Sedhain, S. et al. (2015). “Autorec: Autoencoders meet collaborative filtering”. In: *WWW*.

Seroussi, Yanir, Fabian Bohnert, and Ingrid Zukerman (2011). “Personalised Rating Prediction for New Users Using Latent Factor Models”. In: *ACM conference on Hypertext and hypermedia*, pp. 47–56.

Shi, Y., M. Larson, and A. Hanjalic (2014). “Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges”. In: *CSUR*.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587, pp. 484–489.

Simonetto, A. et al. (2015). “Prediction-Correction Methods for Time-Varying Convex Optimization”. In: *IEEE Asilomar Conference on Signals, Systems and Computers*.

Simons, Taylor and Dah-Jye Lee (2019). “A Review of Binarized Neural Networks”. In: *Electronics* 8.661.

Simonyan, K. and A. Zisserman (2014). “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv*. URL: `arXiv:1409.1556`.

Sindhwani, V., T. Sainath, and S. Kumar (2015). “Structured Transforms for Small-Footprint Deep Learning”. In: *NIPS*, pp. 3088–3096.

Sompolinsky, H., A. Crisanti, and H. Sommers (1988). "Chaos in Random Neural Networks". In: *Physical Review Letters* 61.3, pp. 259–262.

Srinivas, S. and R. Venkatesh Babu (2015). "Data-free Parameter Pruning for Deep Neural Networks". In: *arXiv*. URL: arXiv:1507.06149.

Strub, F., J. Mary, and R. Gaudel (2016). "Hybrid collaborative filtering with autoencoders". In: *arXiv*. URL: arXiv:1603.00806.

Süli, E. and D. Mayers (2003). *An Introduction to Numerical Analysis*. Cambridge University Press, pp. 325–329.

Sun, Kaiming He Xiangyu Zhang Shaoqing Ren Jian (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *arXiv*. URL: arXiv:1502.01852.

Supercomputer with innovative software beats Go Professional (Retrieved 2008-12-19). URL: http://www.cs.unimaas.nl/g.chaslot/muyungwan-mogo/.

Szegedy, C. et al. (2015). "Going Deeper with Convolutions". In: *CVPR*.

The Netflix Prize (Retrieved 2009-9-24). URL: http://www.netflixprize.com/community/viewtopic.php?id=1537.

Tieleman, T. and G. Hinton (2012). "Lecture 6a - rmsprop". In: *COURSERA: Neural Networks for Machine Learning*.

Training LeNet on MNIST with Caffe (Retrieved 2020-07-31). URL: https://caffe.berkeleyvision.org/gathered/examples/mnist.html.

Turing, Alan (1950). "Computing Machinery and Intelligence". In: *The Sun*, pp. 433–460. URL: turing.org.uk.

Vanhoucke, V., A. Senior, and M. Z. Mao (2011). "Improving the Speed of Neural Networks on CPUs". In: *NIPS*.

Veit, Andreas, Michael Wilber, and Serge Belongie (2016). "Residual Networks Behave Like Ensembles of Relatively Shallow Networks". In: *arXiv*. URL: arXiv:1605.06431v2.

Wang, H., N. Wang, and D.-Y. Yeung (2015). "Collaborative deep learning for recommender systems". In: *KDD*.

Wang, H., S. Xingjian, and D.-Y. Yeung (2016). "Collaborative recurrent autoencoder: Recommend while learning to fill in the blanks". In: *NIPS*.

Wen, Wei et al. (2016). "Learning Structured Sparsity in Deep Neural Networks". In: *NIPS*, pp. 2074–2082.

Wu, Y. et al. (2016). “Collaborative denoising auto-encoders for top-n recommender systems”. In: *WSDM*.

Xie, S. et al. (2016). “Aggregated Residual Transformations for Deep Neural Networks”. In: *arXiv*. URL: arXiv:1704.04861.

Yuan, M. and Y. Lin (2006). “Model selection and estimation in regression with grouped variables”. In: *Journal of the Royal Statistical Society. Series B* 68.1, pp. 49–67.

Zeiler, M. D. (2012). “ADADELTA: An Adaptive Learning Rate Method”. In: *arXiv*. URL: arXiv:1212.5701.

Zeiler, M. D. and R. Fergus (2013). “Visualizing and Understanding Convolutional Networks”. In: *arXiv*. URL: arXiv:1311.2901.

Zeng, Wei et al. (2014). “Uncovering the information core in recommender systems”. In: *Scientific Reports*, pp. 6140–6152.

Zhang, F. et al. (2016). “Collaborative knowledge base embedding for recommender systems”. In: *KDD*.

Zhang, S., L. Yao, and A. Sun (2017). “Deep learning based recommender system: A survey and new perspectives”. In: *arXiv*. URL: arXiv:1707.07435.

Zhang, X. et al. (2017). “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices”. In: *arXiv*. URL: arXiv:1707.01083.

Zhang, Xiao, Simon S. Du, and Quanquan Gu (2018). “Fast and Sample Efficient Inductive Matrix Completion via Multi-Phase Procrustes Flow”. In: *ICML*.

Zhang, Y., E. Chuangsuwanich, et al. (2015). “Prediction-Adaptation-Correction Recurrent Neural Networks for Low-Resource Language Speech Recognition”. In: *arXiv*. URL: arXiv:1510.08985.

Zhang, Y., D. Yu, et al. (2015). “Speech recognition with prediction-adaptation-correction recurrent neural networks”. In: *IEEE ICASSP*.