University of New Mexico

UNM Digital Repository

Mathematics & Statistics ETDs

Electronic Theses and Dissertations

Spring 4-4-2023

Multilevel Optimization with Dropout for Neural Networks

Gary Joseph Saavedra

University of New Mexico

Follow this and additional works at: https://digitalrepository.unm.edu/math_etds

Part of the Applied Mathematics Commons, Mathematics Commons, and the Statistics and Probability Commons

Recommended Citation

Saavedra, Gary Joseph. "Multilevel Optimization with Dropout for Neural Networks." (2023). https://digitalrepository.unm.edu/math_etds/199

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at UNM Digital Repository. It has been accepted for inclusion in Mathematics & Statistics ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Candidate

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

, Chairperson

Multilevel Optimization with Dropout for Neural Networks

by

Gary Saavedra

M.S., Computer Science, Georgia Institute of Technology, 2018

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mathematics

The University of New Mexico

Albuquerque, New Mexico

May, 2023

Dedication

Dedicated to my wonderful wife Shasta and beautiful baby girl Iris.

Acknowledgments

I would like to thank my thesis advisor Dr. Jacob Schroder and colleague Dr. Eric Cyr for their support.

Multilevel Optimization with Dropout for Neural Networks

by

Gary Saavedra

M.S., Computer Science, Georgia Institute of Technology, 2018

M.S., Mathematics, University of New Mexico, 2023

Abstract

Large neural networks have become ubiquitous in machine learning. Despite their widespread use, the optimization process for training a neural network remains computationally expensive and does not necessarily create networks that generalize well to unseen data. In addition, the difficulty of training increases as the size of the neural network grows. In this thesis, we introduce the novel MGDrop and SMGDrop algorithms which use a multigrid optimization scheme with a dropout coarsening operator to train neural networks. In contrast to other standard neural network training schemes, MGDrop explicitly utilizes information from smaller sub-networks which act as approximations of the full network. We empirically show that both MGDrop and SMGDrop perform comparably to existing training algorithms and in some cases are able to beat current algorithms in terms of accuracy and training time. In addition, we derive theoretical descriptions of the underlying update rules and their effects on network gradients.

Contents

List of Figures **ix**

List of Tables **xii**

Glossary **xiii**

1 Introduction **1**

2 Related Work **4**

3 Introduction to Machine Learning **5**

3.1 Basics of Machine Learning 5

3.2 Neurons 6

3.3 Neural Networks 8

3.3.1 Training the Neural Network 8

4 Datasets **12**

5 Multigrid Optimization **14**

5.1 MG/OPT algorithm 14

5.2 A Walkthrough of a 2-Level Problem 16

5.3 Comparison of Two-Level MG/OPT to Linear Multigrid Error Correction 17

5.4 MGDrop: MG/OPT with Dropout for Neural Networks 18

6 Dropout as a coarsening mechanism **22**

6.1 Understanding Dropout as a Coarsening Mechanism 22

6.1.1 Coarse Optimization Problem Weight Update 23

6.2 Understanding Updates for the MGDrop Algorithm 24

6.2.1 With Tau Correction 25

6.2.2 Without Tau Correction 27

7 Experimental Results **30**

7.1 Defining a Work Unit 30

7.2 Justification for Dropout as a Coarsening Operator 31

7.3 Performance of Algorithms 32

7.3.1 Performance of MGDrop and SMGDrop for Varied Network Sizes 32

7.3.2 Performance for Different Dropout Rates 33

7.3.3 Performance for Different N_c 33

7.3.4 Performance for Varying Dataset Sizes 34

8 Conclusion 46

References 48

List of Figures

3.1 A simple neuron with input features $x^{(i)}$, weights $w^{(i)}$, bias b, and output y. These neurons form the building blocks of a neural network. 7

3.2 (a) A fully connected neural network with 2 hidden layers and 1 output layer. (b) A neural network with dropout applied. Dropout randomly picks nodes with user-specified probability p and removes the node and all its connections. 9

4.1 Peaks dataset. Each color represents a different class. 12

4.2 MNIST data examples. 13

7.1 The timing for various steps of a nonlinear neural network optimization process for (a) 2 layers, (b) 6 layers, and (c) 10 layers. Each layer contains 8 hidden nodes. The "forward" and "backward" labels refer to a forward pass and backward pass through the network. The "parameter update" refers to the adjustment of the weights using the gradient obtained during the backward pass. The "loss" refers to the calculation of the loss after the forward pass. 35

7.2 Distribution of accuracies with respect to training time measured in work units for 2-layer networks. Each layer has (a) 64, (b) 128, and (c) 256 hidden nodes. The network was trained with a batch size of 4 using the AdaGrad optimizer and learning rate 0.1. Each violin plot is a distribution of accuracies over 100 different dropped out networks with p set to 0.5 in the dropout algorithm. Each accuracy was calculated using 1000 samples. For reference, the full network accuracy is shown in red. We can see that the mean of the distribution of accuracies over different dropout networks (middle blue line) often closely matches the accuracy of the full network (red dot). We can also see that this effect holds throughout the training process. . 36

7.3 Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 128 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds. 37

7.4 Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 512 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds. 38

7.5 Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 1024 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds. 39

7.6 Performance of MGDrop for various dropout rates for 2, 6, and 10 layer networks. Baseline algorithms with the same dropout rates are shown for comparison. 40

7.7 Performance of SMGDrop for various dropout rates for 2, 6, and 10 layer networks. Baseline algorithms with the same dropout rates are shown for comparison. 41

7.8 MGDrop Performance for different number of coarse optimization steps N_c. Each curve represents the performance for networks with different number of layers. Each layer has 128 nodes. 42

7.9 SMGDrop Performance for different number of coarse optimization steps N_c. Each curve represents the performance for networks with different number of layers. Each layer has 128 nodes. 43

7.10 Performance of various algorithms for varying amounts of the Peaks training data. 44

7.11 Performance of various algorithms for varying amounts of the MNIST training data. 45

List of Tables

5.1 Comparison of MG/OPT and Linear Multigrid. 21

Glossary

w	Weight parameters of the neural network
b	Bias parameters of the neural network
x	The data/input to a neural network
p	Probability of removing an individual node with dropout
f	The loss function of the neural network

Chapter 1

Introduction

Large neural networks (NN) have become ubiquitous in machine learning [2, 11, 17, 15, 28, 1]. Despite their widespread use, the optimization process for training a NN remains computationally expensive and does not necessarily create networks that generalize well to unseen data. In addition, the difficulty of training increases as the size of the NN grows.

The choice of optimization algorithm for NN training is crucial and determines both the speed of convergence and final accuracy of the network [24]. Despite the progress in optimization research, it is still common for NN training algorithms to converge to local optima [14]. In addition, the tendency to get stuck in local optima increases as NN sizes increase [13].

In general, there are two ways for an NN to increase in size: 1) by adding more nodes to a layer and/or 2) by adding more layers. Each type of growth creates additional difficulties for the optimization algorithm. First, additional nodes or layers increases the number of parameters which must be considered in the optimization space. Second, with large numbers of layers, gradient information needed for optimization may vanish as it propagates through the layers [5]. This vanishing gradient problem is caused by the multiplication of many small partial derivatives needed for the weight update process [4]. Finally, large models are at increased risk of memorizing training data (overfitting) and hence less likely to generalize well to new data [23].

The choice of activation function for an NN can also strongly influence the risk of vanishing gradients and overfitting [19]. ReLU activation functions [3] in particular have been studied as a means of improving NN training. There are several advantages to using a ReLU function. First, NNs with ReLU activation functions do not experience vanishing gradients since the gradient is always equal to either 0 or 1. Second, ReLU often creates inactive neurons due to zero-value outputs for negative neuron inputs. This phenomenon is known as “dying ReLU” and reduces overfitting via network sparsity. One disadvantage of ReLUs is that neurons may have trouble learning from examples whose inputs are negative. However, when batching multiple examples together this issue becomes less probable [19].

Most NN optimization algorithms utilize some form of stochasticity to improve performance but few leverage smaller sub-networks that act as an approximation to the original large network. Instead, current optimization algorithms attempt to train the full NN which is costly and may lead to overfitting if the network is overparameterized compared to the dataset.

Regularization is another common approach to limit model size/complexity and improve the generalization of the final model. In particular, dropout [23] is a regularization scheme that reduces the number of parameters in a NN model by stochastically removing nodes during training. Hence, dropout improves optimization by utilizing parameter updates from random sub-problems at each training iteration. However, current implementations of dropout do not directly transfer information from problem to sub-problem or vice versa. This limitation may result in suboptimal performance as each iteration of dropout solves random versions of the original problem with no additional information.

Multigrid optimization (MG/OPT) [20] is a multilevel approach which provides an explicit mechanism for transferring information between the original problem and the sub-problem. By utilizing this information sharing, MG/OPT can increase the speed of convergence of the optimization algorithm, as well as the quality of the final solution. In addition, multigrid methods have been shown particularly to improve convergence for very large problems primarily involving partial differential equations [25]. In this thesis we implement MultiGrid Dropout (MGDrop) and Stochastic MultiGrid Dropout (SMGDrop): an MG/OPT scheme for NN training which uses dropout as a way of defining the smaller sub-problems. We show that each of these algorithms performs well compared to their baseline counterparts. We also show that SMGDrop is able to outperform baseline algorithms under certain circumstances.

This thesis is arranged as follows. In Chapter 2 we cover prior work in the space of multilevel NN training. In Chapter 3 we introduce basic machine learning and NN concepts. In Chapter 4 we discuss the data used for our experiments. In Chapter 5 we introduce the MG/OPT algorithm and our extension algorithms, MGDrop and SMGDrop, to NNs. In Chapter 6 we derive theoretical properties for dropout as a coarsening mechanism. Finally, in Chapter 7 we show the results of experiments comparing MGDrop, SMGDrop, and baseline algorithms.

Chapter 2

Related Work

There is very little prior work applying multigrid methods to neural networks. Günther et al. utilize a multigrid in time (MGRIT) [12] scheme to train very deep ResNet models [16]. Cyr et al. also utilize MGRIT for multilevel initialization of ResNet models prior to training [8]. However, both of these prior works differ from our work in several key ways. First, they utilize MGRIT schemes which coarsen whole layers (i.e., they remove entire layers) whereas our approach uses MG/OPT with dropout which coarsens within layers (i.e., removes subsets of nodes within layers). Second, MGRIT methods require model architectures that can be interpreted as an ODE, e.g., a forward Euler discretization. In contrast, our methods put no constraint on the type of model that can be considered.

Planta et al. implement an MG/OPT approach for training NNs and show improved convergence speed and accuracy over stochastic gradient descent [26]. Similarly, Dun et al. implement a multilevel approach without an explicit coarse error correction and show improved parallelization without major degredation of model performance [10]. However, both of these works also restrict their methods to ResNet models and coarsen strictly by removing layers.

Chapter 3

Introduction to Machine Learning

3.1 Basics of Machine Learning

Machine learning is a field which focuses on creating algorithms to learn from data. Machine learning algorithms can be broadly split into two types: *supervised* and *unsupervised*. Supervised algorithms are used to predict a specific label for a data point, whereas unsupervised algorithms are used to explore and describe the properties of the data. Supervised algorithms are further separated into two types of tasks: *classification* and *regression*. Classification algorithms take an input data point and predict a *class* label for that data e.g., given an image of a cat the algorithm will label the image as having the "cat" class. Regression algorithms take an input data point and predict a numerical output value e.g., given housing data it will predict the price of a given home. For an in-depth introduction to machine learning see [7]. In this work we exclusively focus on supervised classification algorithms. Specifically, we focus on neural networks (NN).

Machine learning algorithms operate on datasets. A dataset consists of individual data points x of size n which are described by individual features $x^{(i)}$ where $1 \leq i \leq n$.

The structure of the individual features is highly dependent on the dataset. For instance, in images each individual $x^{(i)}$ is a pixel of the image whereas in text each $x^{(i)}$ may correspond to a single word or character. There are also *tabular* datasets where features are explicitly hand-engineered based on properties of the data, e.g., in a house pricing dataset a feature may correspond to square feet, location, etc.

A supervised model S takes a data point as input and outputs a class label y, i.e.,

$$y = S(x).$$

A supervised model contains a set of internal parameters $w^{(i)}$ which control the output response of the model. These internal parameters are tuned via a *training* process. The specific details of the training process vary widely based on the supervised algorithm and task. In general though, training consists of an iterative optimization process whereby information from the current output, y, is used to adjust internal parameters to improve performance of the model and reduce the number of incorrect predictions.

3.2 Neurons

Before we describe the fully connected NN we must first describe its individual parts. A NN consists of individual neurons. The basic structure of the neuron is shown in Figure 3.1. A neuron performs a weighted summation of inputs i.e., given a set of inputs $x^{(i)}$ and a set of scalar parameters $w^{(i)}$, b where $1 \leq i \leq n$, a neuron computes the output

$$y = w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + ... + w^{(n)}x^{(n)} + b.$$

The $w^{(i)}$ are referred to as the weights of the neuron and the b is referred to as the bias. The act of feeding inputs through a neuron and receiving an output is referred

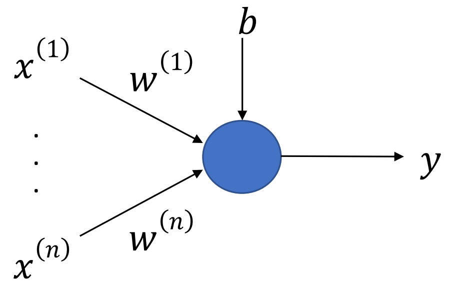


Figure 3.1: A simple neuron with input features $x^{(i)}$, weights $w^{(i)}$, bias b, and output y. These neurons form the building blocks of a neural network.

to as a *forward pass*. The weights are parameters of the neuron and can be tuned via training for the particular task at hand. It is common to feed the weighted summation through a non-linear function σ such that

$$y = \sigma(w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + ... + w^{(n)}x^{(n)} + b).$$

This non-linearity, often referred to as an *activation function*, allows for richer representations and often improves model performance. Common activation functions include the ReLU, Sigmoid, and tanh functions. In this work we utilize the ReLU function given by

$$\sigma_{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} .$$

See [3] for more information on common activation functions used in NNs. A single neuron can be used for binary classification. We say that the neuron has classified a data point as class 0 if y is below a certain user-specified threshold and class 1 if y is above the threshold.

3.3 Neural Networks

Single neurons are limited in the types of tasks they can perform successfully. For more complex tasks we can combine neurons together to form neural networks as shown in Figure 3.2a. Here, a column of neurons forms a *layer* where each neuron outputs a weighted summation of the inputs as shown in the previous section. The output of this layer of neurons is then fed into the next layer of neurons. We can continue this process by adding more layers. Each layer increases the representational power of the NN and hence improves its ability to do classification. However, the increased depth comes with additional computational cost and optimization difficulties. The final layer, or output layer, consists of a set of nodes that perform the classification. We denote this set of nodes as $y^{(i)}$ where $1 \leq i \leq n_c$ and n_c is the number of classes.

A softmax layer [14] is commonly used to convert the final layer outputs into a probability distribution. The layers before the output final layer are often referred to as the *hidden layers*. Overall, we can think of the entire feedforward mechanism as taking in a set of data features, $x^{(i)}$, successively transforming the features at each hidden layer into a more descriptive set of features, $y^{(i)}$, and using these descriptive features for classification in the final layer. We implement all neural network architectures in PyTorch 1.10.2 [21].

3.3.1 Training the Neural Network

In order for the NN to perform a classification task successfully we must tune the neuron weights through *training*. Training is an iterative process that consists of four primary steps: 1) a feedforward step for a set of data points, 2) computing the NN *loss* (i.e., the error) on the data points, 3) assigning blame for the error on each of the individual weights and biases with a gradient computed via *backpropagation*, and

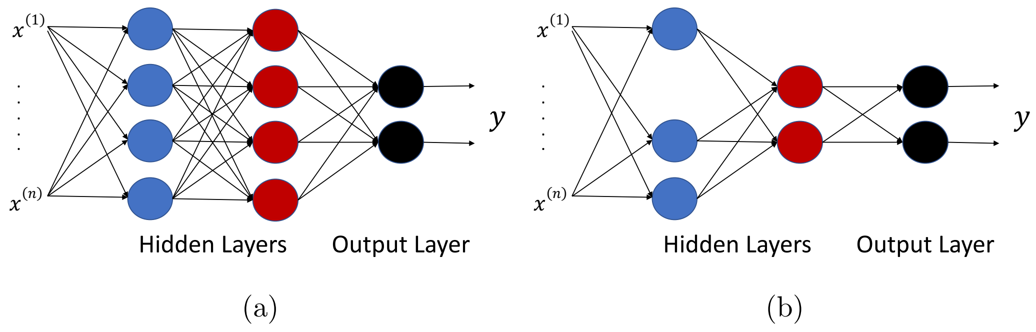


Figure 3.2: (a) A fully connected neural network with 2 hidden layers and 1 output layer. (b) A neural network with dropout applied. Dropout randomly picks nodes with user-specified probability p and removes the node and all its connections.

4) updating the weights and biases to reduce the error. We discussed the feedforward mechanism in previous sections. We will now turn our attention to Steps 2 - 4.

Loss Functions

In order to quantify the performance of an NN we require a measurement of the output error after a feedforward pass. Loss functions measure the difference between expected output and the NN output. There are a wide variety of loss functions with different strengths. For an overview of loss functions see [27]. For this work, we utilize the popular cross entropy loss

$$f_{CE}(x) = -\sum_{i=1}^{n_c} \hat{y}^{(i)} \log\left(p^{(i)}(x)\right)$$

where $\hat{y}^{(i)}$ is the ground truth class label for class i and $p^{(i)}$ is the softmax output probability for class i and is given by

$$p^{(i)} = \frac{e^{y^{(i)}}}{\sum_j^{n_c} e^{y^{(j)}}}.$$

Backpropagation

After computing the loss we must quantify how much each NN parameter contributed to that loss. Backpropagation is the most commonly used mechanism. Backpropagation computes the gradient, $\frac{\partial f(x)}{\partial w^{(i)}}$, of every NN parameter, $w^{(i)}$, with respect to the loss by conducting a *backward pass.* During the backward pass the chain rule is utilized to calculate gradients along each path of the network. After a full backward pass each weight and bias of the network has an associated gradient which is used to determine how much that particular parameter contributed to the loss. See [22] for a more in depth explanation of backpropagation.

Weight Updates via Optimization

After computing the gradients of each parameter via backpropagation we update the parameters of the model. A wide variety of update rules exist. The most basic is the gradient descent update rule

$$w^{(i)} \leftarrow w^{(i)} - \beta \frac{\partial f(x)}{\partial w^{(i)}},$$

where β is a user-specified learning rate and $\frac{\partial f(x)}{\partial w^{(i)}}$ is the gradient of the loss function $f(x)$ with respect to parameter $w^{(i)}$. Gradient descent computes $\frac{\partial f(x)}{\partial w^{(i)}}$ based on the entire training set. A more common implementation known as stochastic gradient descent (SGD) computes $\frac{\partial f(x)}{\partial w^{(i)}}$ based on only a subset of the training set.

Introduction to Dropout

Dropout [23] is a popular regularization technique used to prevent overfitting in neural networks. With dropout on a fully connected layer, nodes are randomly removed from the network according to a user specified probability p as shown in Figure 3.2b. In this paper, we use dropout as a stochastic coarsening mechanism where the coarse grid is a copy of the fine grid neural network with nodes randomly dropped out.

Chapter 4

Datasets

We utilize the Peaks dataset developed in [18]. Each data point is classified as 1 of 5 classes and contains two features: an x and y coordinate. Peaks consists of 5000 training examples with 1000 examples per class. See Figure 4.1 for a visual representation (cf. Figure 6 from [18].

We also utilize the MNIST dataset [9] which is a commonly used dataset for benchmarking machine learning algorithms for image classification. MNIST consists

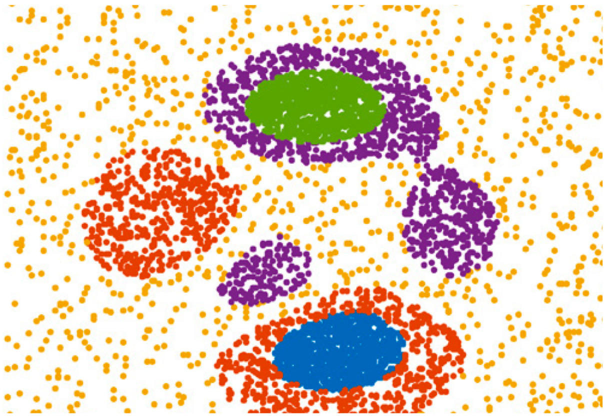


Figure 4.1: Peaks dataset. Each color represents a different class.

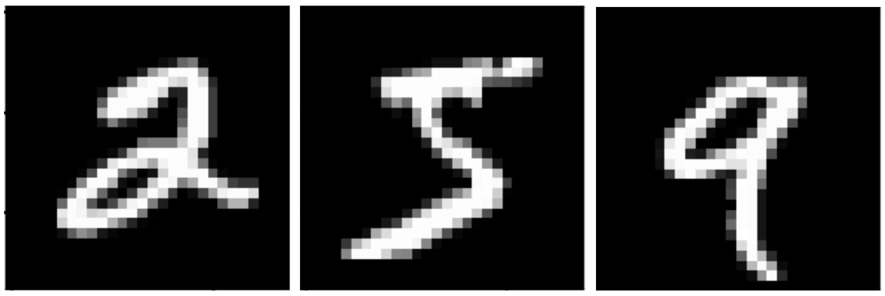


Figure 4.2: MNIST data examples.

of 28x28 pixel image data of handwritten digits 0-9. There are 60,000 total training examples with 6000 examples per class. The validation set consists of 10,000 examples with 1000 examples per class. See Figure 4.2 for a visual representation.

Chapter 5

Multigrid Optimization

5.1 MG/OPT algorithm

In this section we introduce the MG/OPT algorithm [20] and a novel application to neural networks. We use w to indicate the solution output by MG/OPT. We use bar notation (e.g. $\bar{w}$) to indicate solutions at the next coarse level. The coarse level is a sparse version of the current level. Typically the problem size is cut in half at the coarse level. Additionally, we use h to indicate the current level and H to indicate the next coarse level. We denote the following operators (defined in more detail later):

f_h :	objective function at level h
I_h^H :	Transfer matrix from the fine level to the coarse level
I_H^h :	Transfer matrix from the coarse level to the fine level.

Given the initial guess w_0, a single iteration of the MG/OPT algorithm is defined in [20] as follows (note that this is a recursive algorithm).

Algorithm $MG/OPT(w_0, f_h)$:

1. If this is the coarsest grid solve

$$\underset{w}{\text{minimize}} \quad f_h(w)$$

   using w_0 as initial guess and return result w

2. Otherwise, apply N_0 iterations of an optimization algorithm to the original problem using w_0 as the initial guess to obtain $w_{(1)}$.

3. Compute

$$\begin{aligned} \bar{w}_{(1)} &= I_h^H w_{(1)} \\ g_{(1)} &= \nabla f_h(w_{(1)}) \\ \bar{g}_{(1)} &= \nabla f_H(\bar{w}_{(1)}) \\ \bar{v} &= \bar{g}_{(1)} - I_h^H g_{(1)} \end{aligned}$$

4. Apply the multigrid method recursively, with initial guess $\bar{w}_{(1)}$ to

$$\underset{\bar{w}}{\text{minimize}} \quad f_H(\bar{w}) - \bar{v}^T \bar{w}$$

   and let $\bar{w}_{(2)}$ be the result. That is

$$\bar{w}_{(2)} = MG/OPT(\bar{w}_{(1)}, f_H(\bar{w}) - \bar{v}^T \bar{w})$$

5. Compute error approximation

$$e_{(2)} = I_H^h \big(\bar{w}_{(2)} - \bar{w}_{(1)}\big)$$

6. Perform a line search to obtain

$$w_{(2)} \leftarrow w_{(1)} + \alpha e_{(2)}$$

7. Apply N_1 iterations of an optimization algorithm to the original problem, with initial guess $w_{(2)}$ to obtain $w_{(3)}$. Return $w_{(3)}$.

We note that the $\bar{v}^T \bar{w}$ term is often referred to as the tau correction term.

5.2 A Walkthrough of a 2-Level Problem

This section is intended to show how MG/OPT would iterate from start to finish in a two level setting. We refer to the steps of the MG/OPT algorithm above by using italicized notation e.g., *Step number.*

1. On the first pass through we are on the fine level, so we bypass *Step 1* of MG/OPT.

2. We follow *Step 2* and apply N_0 iterations of an optimization algorithm to f_h. This gives a new solution on the fine grid, $w_{(1)}$.

3. Next we compute the quantities specified in *Step 3*. Here we obtain the following: $\bar{w}_{(1)}$: the coarsened solution of the fine grid. Note that this is obtained by taking the fine grid solution and applying the restriction operator I_h^H. It is not obtained by solving the optimization problem on the coarse level.
   $g_{(1)}$: The gradient on the fine level.
   $\bar{g}_{(1)}$: The gradient on the coarse level.
   $\bar{v}$: The difference between the coarse gradient and the coarsened version of the fine grid gradient.

4. We now apply the multigrid method, with initial guess $\bar{w}_{(1)}$ to the coarse level

$$\underset{\bar{w}}{\text{minimize}} \quad f_H(\bar{w}) - \bar{v}^T\bar{w}$$

   and let $\bar{w}_{(2)}$ be the result. This recursive call returns us to *Step 1* where we solve the optimization problem on the coarse level. If we were using more than 2 levels then we would once again skip *Step 1* and repeat the above steps until we reached the coarsest level. We can think of $\bar{w}_{(2)}$ as a correction to $\bar{w}_{(1)}$. The $\bar{v}^T\bar{w}$ in the optimization problem gives a sense of the difference in the algorithm step direction/magnitude between the fine and coarse grid. This is the amount of disagreement that is created when we try to minimize the coarse problem instead of the fine problem. Then, by subtracting this term from the optimization problem, we remove this source of disagreement. This allows the optimization algorithm to discover a better solution on the coarse grid.

5. Now that we have the $\bar{w}_{(2)}$ we can compute $e_{(2)}$ from *Step 5* which is an approximate measure of the error on the fine grid. This is obtained by prolongating $\bar{w}_{(2)} - \bar{w}_{(1)}$, with I_H^h.

6. Follow *Step 6* and update the solution on the fine grid.

7. Repeat until you reach some tolerance.

5.3 Comparison of Two-Level MG/OPT to Linear Multigrid Error Correction

In this section we convert the MG/OPT algorithm to its linear algebra counterpart and compare to the steps of the classic linear multigrid error correction. Our goal is to better understand the algorithm. We use the two grid correction scheme for linear problems in [6]. Here we wish to solve the quadratic optimization problem

$$\underset{w}{\text{minimize}} \quad w^T A w - w^T b$$

which is equivalent to solving the linear system

$$Aw = b.$$

Table 5.1 compares the steps of MG/OPT vs. linear multigrid. Note that these steps (in particular the cancellation of the b terms in Step 3) assume that our restriction operator I_h^H is injection (i.e., it chooses every other fine point to be the set of coarse points). We use the same notation as in Section 5.1 and the following additional notation:

$\bar{A}$:	The construction of the A matrix on the coarse level e.g. a finite difference matrix of size $\frac{n}{2}$ where n is the size of the fine grid. Note that $\bar{A} \neq I_h^H A$.
$\bar{w}$:	The coarse version of vector w, i.e., $I_h^H w$.
$\bar{b}$:	The coarse version of vector b, i.e., $I_h^H b$.

5.4 MGDrop: MG/OPT with Dropout for Neural Networks

In this section we define MGDrop: a novel algorithm for training neural networks using MG/OPT with dropout as a coarsening mechanism. We define the multigrid operators as:

f_h :	NN loss function as defined in Section 3.3.1
∇f_h :	the gradient of the loss function with respect to the NN parameters
I_h^H :	NN dropout with probability p specified by user.
I_H^h :	Injection of coarse grid weights into fine NN
w :	The weights and biases of the NN.

Bar notation represents the weights and biases of the coarse NN. An iteration of optimization consists of the following steps: 1) forward pass, 2) compute loss $f_h(w)$, 3) backward pass, and 4) update weights w of the network via stochastic gradient descent as defined in Section 3.3.1. Given the initial network w_0, we define MGDrop as follows.

Algorithm $MGDrop(w_0, f_h)$:

1. If this is the coarsest grid perform N_c iterations of optimization on f_h and return the updated w.
2. Otherwise, apply N_0 iterations of optimization to the problem f_h to obtain $w_{(1)}$.
3. Compute
$$\begin{aligned} \bar{w}_{(1)} &= I_h^H w_{(1)} \\ g_{(1)} &= \nabla f_h(w_{(1)}) \\ \bar{g}_{(1)} &= \nabla f_H(\bar{w}_{(1)}) \\ \bar{v} &= \bar{g}_{(1)} - I_h^H g_{(1)} \end{aligned}$$
4. Apply the multigrid method recursively.$\bar{w}_{(2)} = MGDrop(\bar{w}_{(1)}, f_H(\bar{w}_{(1)}) - \bar{v}^T \bar{w}_{(1)})$.
5. Compute
$$e_{(2)} = I_H^h \big(\bar{w}_{(2)} - \bar{w}_{(1)}\big)$$
6. Perform a line search to obtain
$$w_{(2)} \leftarrow w_{(1)} + \alpha e_{(2)}$$
7. Apply N_1 iterations of optimization to the original problem f_h, with initial guess $w_{(2)}$ to obtain $w_{(3)}$. Return $w_{(3)}$.

Note that in contrast to the original MG/OPT algorithm, MGDrop does not necessarily fully minimize in Steps 1 and 4. Rather we choose a certain number of optimization iterations to perform.

In addition to MGDrop we consider a stochastic variant of the algorithm which we refer to as SMGDrop. MGDrop uses the same batch of data for each step of the algorithm i.e., a new batch is not used until a full iteration of MGDrop is complete. In contrast, SMGDrop uses a unique batch of data for each unique optimization step within the MGDrop iteration i.e., Step 1, 2, and 7 each use a unique data batch. By considering different data batches, SMGDrop is less likely to overfit than MGDrop.

Step number	MG/OPT step	Linear Multigrid step
Step 2	minimize $\quad w^T A w - w^T b$ return w_1	Relax $Aw = b$ return w_1
Step 3	$\bar{w}_1 = I_h^H w_1$ $g_1 = \nabla f_h(w_1) = Aw_1 - b$ $\bar{g}_1 = \nabla f_H(\bar{w}_1) = \bar{A}\bar{w}_1 - \bar{b}$ $\bar{v} = \bar{g}_1 - I_h^H g_1 = \bar{A}\bar{w}_1 - I_h^H A w_1$	Compute fine and coarse grid residual $-r_1 = Aw_1 - b$ $-\bar{r}_1 = \bar{A}\bar{w}_1 - \bar{b}$
Step 4	minimize $\quad \bar{w}^T \bar{A}\bar{w} - \bar{w}^T \bar{b} - \bar{v}^T \bar{w}$ return $\bar{w}_2$	Solve $\bar{A}\bar{w} = \bar{b} + \bar{v}$ return $\bar{w}_2$ Note that this is equivalent to $\bar{A}\bar{e} = \bar{r}_1$ where $\bar{e} = \bar{w} - \bar{w}_1$ and $\bar{r}_1 = I_h^H r_1 = I_h^H (b - Aw_1)$
Step 5	Compute $e_2 = I_H^h \Big(\bar{w}_2 - \bar{w}_1\Big)$	Interpolate $e = I_H^h \bar{e}$ which is equivalent to $e = I_H^h \Big(\bar{w}_2 - \bar{w}_1\Big)$
Step 6	Perform a line search to obtain $w_2 \leftarrow w_1 + \alpha e_2$	Correct fine grid approximation $w_2 \leftarrow w_1 + e$
Step 7	minimize $\quad w^T A w - w^T b$ with initial guess w_2 return w_3	Relax $Aw = b$ with initial guess w_2 return w_3

Table 5.1: Comparison of MG/OPT and Linear Multigrid.

Chapter 6

Dropout as a coarsening mechanism

6.1 Understanding Dropout as a Coarsening Mechanism

We wish to understand how the MGDrop algorithm affects weight updates. For the following derivation we follow the analysis of [14] (Ch. 7, pg 227). Consider a loss function $f(w)$ where w are the NN weights. Further, consider a small neighborhood such that we can make a quadratic approximation of this loss function with optimal weights given by $w^* = \arg\min_w f(w)$. Then using a Taylor series expansion, we can form the approximation

$$\hat{f}(w) = f(w^*) + \frac{1}{2}(w - w^*)^T \mathbf{H}(w - w^*)$$

where $\mathbf{H}$ is the Hessian of the loss function f. Then the gradient is given by

$$\nabla_w \hat{f}(w) = \mathbf{H}(w - w^*), \tag{6.1}$$

where the minimum occurs when $\nabla_w \hat{f}(w)$ is equal to 0.

6.1.1 Coarse Optimization Problem Weight Update

In this section, we look at the weight updates of the coarse optimization problem. We assume a quadratic approximation in a small neighborhood as in Section 6.1. We adopt the notation from Section 5.1 (i.e., bar notation and capital H subscripts signify that a quantity is associated with the coarse grid; lower case h subscripts and vectors without a bar are fine grid quantities; numbered subscripts represent a quantity derived at a particular step of the MGOPT algorithm). The coarse optimization problem of MG/OPT is given by

$$\underset{\bar{w}}{\text{minimize}} \quad f_H(\bar{w}) - \bar{v}^T\bar{w} \tag{6.2}$$

where

$$\begin{aligned}\bar{v} &= \nabla f_H(\bar{w}_{(1)}) - I_h^H \nabla f_h(w_{(1)})\\ \bar{w}_{(1)} &= I_h^H w_{(1)}\end{aligned}$$

and I_h^H is the dropout operator. Taking the gradient of the objective function (6.2) and approximating with a Taylor series as in (6.1) gives

$$\nabla_w \hat{f}_H(\bar{w}) = \bar{\mathbf{H}}(\bar{w} - \bar{w}^*) - \bar{v}, \tag{6.3}$$

where $\bar{w}^* = \arg\min_{\bar{w}} f_H(\bar{w})$. Then the minimum of the coarse optimization problem (6.3) occurs at $\nabla_w \hat{f}_H(\bar{w}) = 0$ which allows us to write

$$\bar{\mathbf{H}}(\bar{w}^*_{(2)} - \bar{w}^*) - \bar{v} = 0 \tag{6.4}$$

$$\Rightarrow \quad \bar{w}^*_{(2)} = \bar{w}^* + \bar{\mathbf{H}}^{-1}\bar{v} \tag{6.5}$$

where $\bar{w}^*_{(2)}$ corresponds to the NN weights that minimize $\hat{f}(\bar{w})$. Since $\mathbf{H}$ is a positive semi-definite matrix there exists an eigendecomposition such that $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$ where $\mathbf{Q}$ are the eigenvectors of $\mathbf{H}$ and $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues. Thus using the fact that $\mathbf{H}^{-1} = \mathbf{Q}\mathbf{\Lambda}^{-1}\mathbf{Q}^T$, we can write

$$\bar{w}^*_{(2)} = \bar{w}^* + \bar{\mathbf{Q}}\bar{\mathbf{\Lambda}}^{-1}\bar{\mathbf{Q}}^T\bar{v} \tag{6.6}$$

Equation (6.6) shows the optimal solution to the coarse optimization problem consists of the optimal solution of the original coarsened/dropped-out NN ($\bar{w}^*$) plus a scaled $\bar{v}$ term. The $\bar{\mathbf{Q}}\bar{\mathbf{\Lambda}}^{-1}\bar{\mathbf{Q}}^T\bar{v}$ scales each component of $\bar{v}$ in the direction of the eigenvectors of $\mathbf{H}$. Thus, the effect of each of the components of $\bar{v}$ on the weight update will be influenced by the size of the respective eigenvalues. In this case, the influence is proportional to the inverse of the eigenvalues of $\mathbf{H}$, i.e., directions with smaller eigenvalues have more influence on the weight updates. Since large eigenvalues of $\mathbf{H}$ indicate steeper regions of the loss function and smaller eigenvalues indicate less steep regions, the coarse optimization favors weight updates in the less steep direction of the loss. Gradient descent typically concentrates updates on steep regions. Thus, gradient descent in MGDrop primarily updates in steep regions on the fine level and in flatter regions on the coarse level. We hypothesize this could give MGDrop an advantage over stand-alone gradient descent.

6.2 Understanding Updates for the MGDrop Algorithm

We saw in Section 6.1 that the MGDrop error correction mechanism for a simple quadratic problem is heavily influenced by the eigenvalues of the Hessian of the loss. In this section, we wish to understand the correction from the coarse level optimization problem both with and without the tau correction term $\bar{v}^T\bar{w}$. We consider the MGDrop algorithm with 2 levels, no line search, injection as the interpolation operator, and a single optimization step each time the fine and coarse optimization problems are iterated (Steps 1, 4, and 7). Further, for optimization we utilize the gradient descent rule

$$w \leftarrow w - \beta\nabla f(w)$$

where β is the learning rate and f is the loss function of the NN. Further, we represent the fine and coarse components of the gradient $g_{(n)}$ as

$$g_{(n)} = \begin{bmatrix} g^f_{(n)} \\ g^c_{(n)} \end{bmatrix}$$

where $g^c_{(n)}$ are the gradients for the nodes also present on the coarse grid and $g^f_{(n)}$ are the gradients on the fine grid nodes that are not part of the coarse grid. Note that $g^c_{(n)} \neq \bar{g}_{(n)}$. The $\bar{g}_{(n)}$ are computed from an example that has been propagated through a dropped-out (coarse) network whereas $g^c_{(n)}$ are restricted from an example that has been propagated through the full (fine) network. We assume this ordering of fine and coarse grid gradients going forward.

6.2.1 With Tau Correction

In this section, we analyze the effect of including the tau correction term. We begin with an initial NN where $w_{(0)}$ represents the edge weights and node biases. In Step 1 of MGDrop we execute a single fine grid optimization. This step generates the gradients $g_{(0)} = \nabla f(w_{(0)})$ and returns new NN parameters $w_{(1)}$ where

$$\begin{aligned} w_{(1)} &= w_{(0)} - \beta \nabla g_{(0)} \\ &= w_{(0)} - \beta \begin{bmatrix} g^f_{(0)} \\ g^c_{(0)} \end{bmatrix} \end{aligned}$$

Next, $\bar{v}$ is computed, and we execute a single iteration of the coarse grid optimization with initial guess $\bar{w}_{(1)}$ which returns $\bar{w}_{(2)}$ where

$$\begin{aligned} \bar{w}_{(2)} &= \bar{w}_{(1)} - \beta \nabla \big(f_H(\bar{w}_{(1)}) - \bar{v}^T \bar{w}_{(1)} \big) \\ &= \bar{w}_{(1)} - \beta \big(\nabla f_H(\bar{w}_{(1)}) - \bar{v}^T \big) \\ &= \bar{w}_{(1)} - \beta \big(\bar{g}_{(1)} - (\bar{g}_{(1)} - I^H_h g_{(1)}) \big) \\ &= \bar{w}_{(1)} - \beta I^H_h g_{(1)} \end{aligned}$$

Thus, we have that

$$\begin{aligned} e_2 &= I_H^h\big(\bar{w}_1 - \beta I_h^H g_{(1)} - \bar{w}_1\big) \\ &= -\beta I_H^h I_h^H g_{(1)} \\ &= -\beta \begin{bmatrix} 0 \\ g^c_{(1)} \end{bmatrix} \end{aligned}$$

where the last equality follows from the fact that the I_h^H operator zeroes out non-coarse gradients and the I_H^h operator is only an injection. Finally with no line search we have that $\alpha = 1$ and our update from the coarse to fine grid becomes

$$\begin{aligned} w_{(2)} &= w_{(1)} + e_{(2)} \\ &= w_{(1)} - \beta \begin{bmatrix} 0 \\ g^c_{(1)} \end{bmatrix} \\ &= w_{(0)} - \beta \begin{bmatrix} g^f_{(0)} \\ g^c_{(0)} \end{bmatrix} - \beta \begin{bmatrix} 0 \\ g^c_{(1)} \end{bmatrix} \end{aligned}$$

Notice that the gradients derived from the coarse optimization problem ($\bar{g}_{(1)}$) do not appear anywhere in this update rule. Thus, the dropped out network is actually playing no role in the NN update. Instead the update to our original network ($w_{(0)}$) consists of the gradients from the initial fine grid optimization step ($g_{(0)}$) and the coarse-node gradients from a second fine grid step ($g^c_{(1)}$). In essence, with the assumptions on MGDrop specified at the beginning of this section, the algorithm can be simplified to

1. Run a fine grid forward/backward to compute gradients
2. Update all weights of the NN
3. Run a second fine grid forward/backward to compute gradients

4. Randomly select nodes with probability p and update the NN only on the selected nodes

5. Run a third fine grid forward/backward to compute gradients

6. Update all weights of the NN

7. Repeat

Explicitly coding MGDrop as this simplified algorithm would greatly reduce the work units required. However, for this thesis we do not use the simplified algorithm in our experiments.

If we instead execute n iterations on the coarse grid in Step 4, then we obtain the following update:

$$\bar{w}_{(n+1)} = \bar{w}_{(1)} - n\beta\big(I_h^H g_{(1)} - \bar{g}_{(1)}\big) - \beta \sum_{n=1}^{n} \bar{g}_{(i)}$$

and hence

$$e_2 = -n\beta I_H^h\big(I_h^H g_{(1)} - \bar{g}_{(1)}\big) - \beta I_H^h \sum_{n=1}^{n} \bar{g}_{(i)}$$

where $\bar{g}_{(i)} = \nabla f_H(\bar{w}_{(i)})$. Here we see that the gradients from the coarse optimization problem do affect the final update. Thus, in order to include the effect of coarse grid gradients we require at least 2 coarse level optimization iterations.

6.2.2 Without Tau Correction

In this section we analyze the effect of excluding the tau correction term. As before, we begin with an initial NN where $w_{(0)}$ represents the weights and node biases. In

Step 1 of MGDrop we execute a single fine grid optimization. This step generates the gradients $g_{(0)} = \nabla f(w_{(0)})$ and returns new NN parameters $w_{(1)}$ where

$$w_{(1)} = w_{(0)} - \beta \begin{bmatrix} g^f_{(0)} \\ g^c_{(0)} \end{bmatrix}$$

We now execute a single iteration of the coarse grid optimization with initial guess $w_{(1)}$ which returns $\bar{w}_{(2)}$ where

$$\begin{aligned} \bar{w}_{(2)} &= \bar{w}_{(1)} - \beta \nabla f_H(\bar{w}_{(1)}) \\ &= \bar{w}_{(1)} - \beta \bar{g}_{(1)} \end{aligned}$$

Thus, we have that

$$\begin{aligned} e_2 &= I^h_H \big(\bar{w}_1 - \beta \bar{g}_{(1)} - \bar{w}_1 \big) \\ &= -\beta I^h_H \bar{g}_{(1)} \end{aligned}$$

Note that $\bar{g}_{(1)}$ is non-zero only for coarse grid nodes and since I^h_H is an injection operator, the updates from e_2 affect only coarse-grid nodes. Finally with no line search we have that $\alpha = 1$ and our update from the coarse to fine grid becomes

$$\begin{aligned} w_{(2)} &= w_{(1)} + e_{(2)} \\ &= w_{(1)} - \beta I^h_H \bar{g}_{(1)} \\ &= w_{(0)} - \beta g_{(0)} - \beta I^h_H \bar{g}_{(1)}. \end{aligned}$$

Notice that without the tau correction our update rule is dependent on the gradients derived from the coarse optimization problem. Thus, this situation utilizes information from a dropped-out network, similar to the way a standard dropout algorithm does, as well as a fine non-dropout network. We can derive a simpler algorithm from the information above. The MGDrop algorithm with no tau correction is identical to the following simpler algorithm

1. Run a fine grid forward/backward to compute gradients
2. Update all weights of the NN
3. Create a coarse NN
4. Run a coarse grid forward/backward to compute gradients
5. Update the NN using the interpolated gradients derived from coarse grid
6. Run a third fine grid forward/backward to compute gradients
7. Update all weights of the NN
8. Repeat

In essence this algorithm consists of a standard iteration, a dropout iteration, and a final standard iteration. This differs from typical dropout training where dropout is applied at every iteration. If we instead execute n coarse grid steps then we obtain the following update:

$$\bar{w}_{(n+1)} = \bar{w}_{(1)} - \beta \sum_{n=1}^{n} \bar{g}_{(i)}$$

and hence

$$e_2 = -\beta I_H^h \sum_{n=1}^{n} \bar{g}_{(i)}$$

which is equivalent to running a standard iteration, n dropout iterations, and a final standard iteration. Note that in this thesis we focus on MGDrop with a tau correction for our experiments.

Chapter 7

Experimental Results

7.1 Defining a Work Unit

In the subsequent results we will be comparing the speed of convergence of MGDrop to several baseline algorithms. In this section we develop a work unit to comparatively benchmark the various algorithms. In general, wall clock time will not suffice for our purposes. PyTorch represents NN parameters as matrices with various operations such as forward/backward passes operating on these matrices. Thus, the wall clock time is highly dependent on the size of the parameter matrices. However, the current implementation of PyTorch dropout only zeroes out entries of the parameter matrix and saves no computation. Thus, wall clock times for a NN with and without dropout are comparable as can be seen in Figure 7.1. Note that for a typical multigrid application it is often the case that the coarse grid problem is computationally cheaper than the fine grid. It may be possible to alter the underlying PyTorch code so that dropout reduces parameter matrix sizes and hence makes the coarse grid problem cheaper. However, such an implementation is beyond the scope of this thesis.

Number of algorithm iterations is also a flawed metric. A single iteration of MG-Drop contains at least 3 forward/backward passes per iteration whereas the baseline algorithms contain only 1 forward/backward pass per iteration. Thus, using number of iterations would give an unfair advantage to MGDrop.

Instead we use a single forward and backward pass to define our work unit. The wall clock time of a forward and backward pass for various network sizes can be seen in Figure 7.1. We see that, in general, the forward/backward passes make up the bulk of the algorithm time and are comparable in magnitude to each other. Thus, we adopt the convention that a forward pass is equivalent to 1 work unit and a backward pass is equivalent to 1 work unit. In addition, as we noted above, dropout operations in Pytorch do not reduce computation time. Thus, we do not reduce the magnitude of our work units as the network is coarsened i.e., a forward or backward pass is equal to 1 unit no matter which grid level we are on.

7.2 Justification for Dropout as a Coarsening Operator

An assumption of multigrid methods is that the coarse grid represents a reasonable approximation of the fine grid [6]. Thus, in order for dropout to function as a coarsening operator we make the well-founded assumption that a dropped out network forms a reasonable approximation of the original network. We provide empirical evidence for this fact in Figure 7.2. Here we measure the distribution of accuracies over different dropped out networks at different points in the training process. By observing the distribution of accuracies over different dropped out networks we can get a sense of how well the dropped out network approximates the original network. We can see in the Figure that for various network sizes and various points in the training process that the accuracy of dropout networks is close to the accuracy of the full network. Thus, we assume that dropout may function as a reasonable coarsening operator.

7.3 Performance of Algorithms

In this section we compare the performance of the MGDrop and SMGDrop algorithms against two baseline algorithms. Our first baseline algorithm is a standard neural network trained with Stochastic Gradient Descent (SGD) and no dropout, and the second baseline algorithm uses SGD with dropout of 0.5. We use a learning rate of 0.01 with no decay and a batch size of 1. In all our results we show the performance of the validation set. Unless stated otherwise, both MGDrop and SMGDrop use dropout rates of 0.0 and 0.5 on the fine and coarse level respectively.

7.3.1 Performance of MGDrop and SMGDrop for Varied Network Sizes

In Figure 7.3 we show the performance of the two baseline algorithms, as well as MGDrop and SMGDrop for networks with 128 hidden nodes per layer. The baseline algorithm shows reasonable performance for most numbers of layers with deeper networks taking more work units to train. The only exception is the network with 16 layers which shows no improvement in performance. Typically deeper networks take longer to train, thus it is likely that more training would improve performance. The baseline algorithm with dropout also shows strong performance for small number of layers. However, the overall performance is worse with dropout especially for larger numbers of layers.

We now turn our attention to MGDrop and SMGDrop in Figure 7.3. Both algorithms show consistently strong performance, with each beating baseline with dropout for any number of layers. SMGDrop typically performs better than MGDrop by converging to higher accuracy more quickly. These results show that utilizing multiple data batches throughout the MG/OPT iteration has a positive effect on performance. The performance of SMGDrop is also comparable to baseline with no dropout. However, baseline with no dropout has significantly higher accuracy than MGDrop.

Figures 7.4 and 7.5 shows performance of the four algorithms for networks with 512 and 1024 nodes per layer respectively. With more nodes per layer MGDrop and SMGDrop beat the baseline algorithms more consistently. Thus, overall we see that MGDrop and SMGDrop tend to perform better with wider or deeper networks.

7.3.2 Performance for Different Dropout Rates

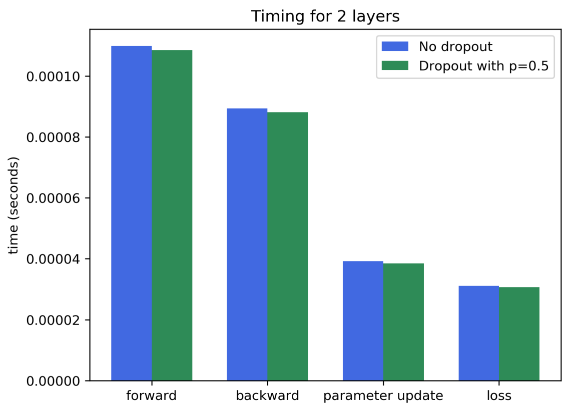
We show the performance of MGDrop and SMGDrop for different dropout rates in Figures 7.6 and 7.7 respectively. Here the dropout rate signifies the amount of dropout on the fine level. The amount of dropout on the coarse level is set to be halfway between 1.0 and the dropout value on the fine level. In general, higher fine level dropout tends to harm performance of both MGDrop and SMGDrop. However, increased dropout also harms the baseline algorithms. This may be an indication that dropout is not beneficial on the Peaks dataset especially when applied on the fine level.

7.3.3 Performance for Different N_c

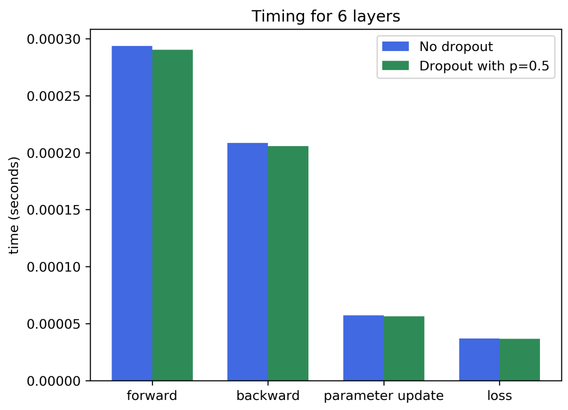
In this section we study the performance of MGDrop and SMGDrop for various numbers of coarse grid iterations N_c. In Figure 7.8 and 7.9 we show the performance of MGDrop and SMGDrop respectively. Each plot in the figure shows the performance for 1, 2, and 3 coarse iterations. Overall, we see that increasing the number of coarse iterations only reduces the accuracy of the model with respect to work units. Further, even with additional work units (100,000 and 120,000 for 2 and 3 coarse iterations respectively) the accuracy is never recovered.

7.3.4 Performance for Varying Dataset Sizes

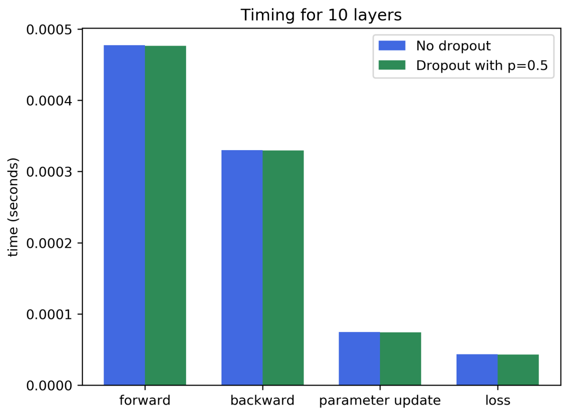
In general, dropout is useful for reducing overfitting of a model to the training data. Generally, overfitting is more likely when the training data has few points. Thus, in this section we study the effect of reducing the training dataset size. In Figure 7.10 and Figure 7.11 we see the performance of the four algorithms for varying sizes of the Peaks and MNIST data respectively. We can see that in several cases SMGDrop now outperforms both baseline algorithms. Thus, SMGDrop may have an advantage over traditional dropout when the training dataset size is small.



(a)

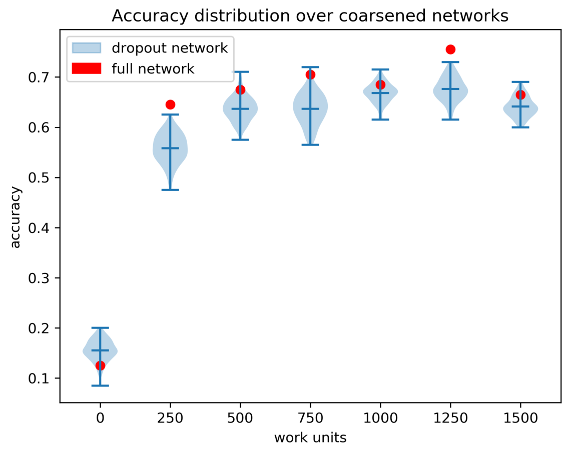


(b)

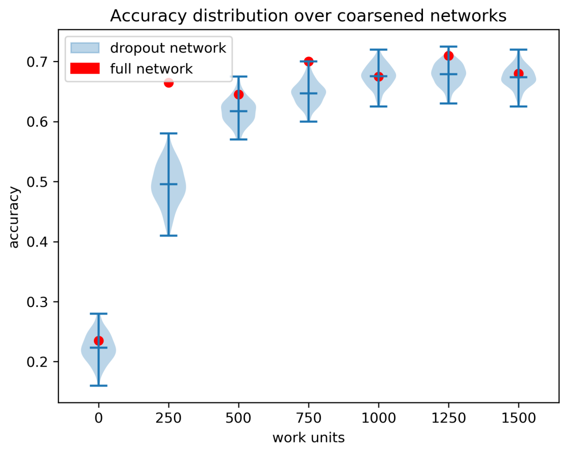


(c)

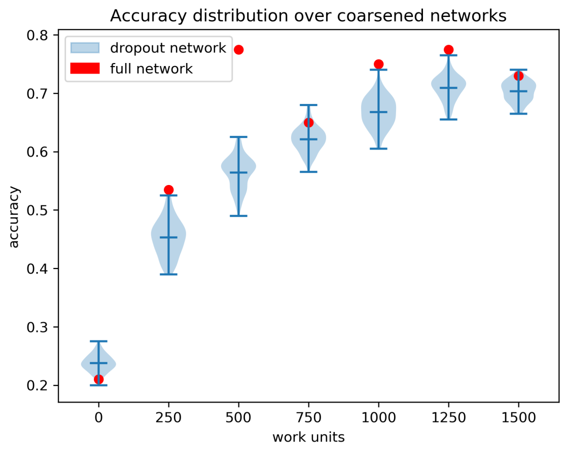
Figure 7.1: The timing for various steps of a nonlinear neural network optimization process for (a) 2 layers, (b) 6 layers, and (c) 10 layers. Each layer contains 8 hidden nodes. The "forward" and "backward" labels refer to a forward pass and backward pass through the network. The "parameter update" refers to the adjustment of the weights using the gradient obtained during the backward pass. The "loss" refers to the calculation of the loss after the forward pass.



(a)

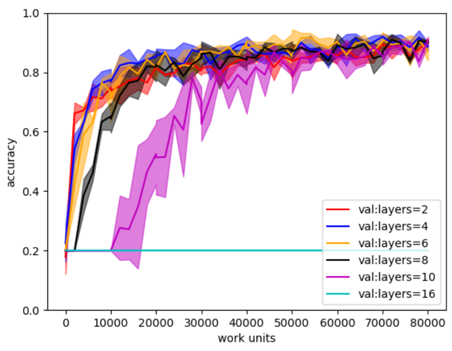


(b)

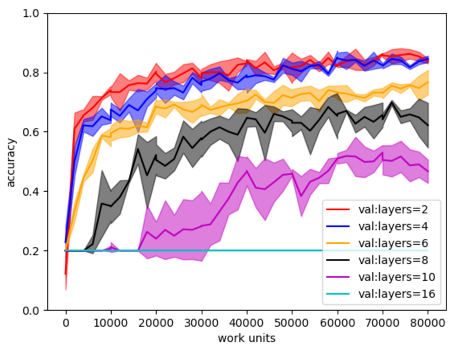


(c)

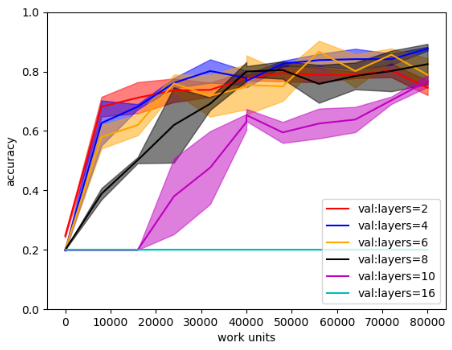
Figure 7.2: Distribution of accuracies with respect to training time measured in work units for 2-layer networks. Each layer has (a) 64, (b) 128, and (c) 256 hidden nodes. The network was trained with a batch size of 4 using the AdaGrad optimizer and learning rate 0.1. Each violin plot is a distribution of accuracies over 100 different dropped out networks with p set to 0.5 in the dropout algorithm. Each accuracy was calculated using 1000 samples. For reference, the full network accuracy is shown in red. We can see that the mean of the distribution of accuracies over different dropout networks (middle blue line) often closely matches the accuracy of the full network (red dot). We can also see that this effect holds throughout the training process.



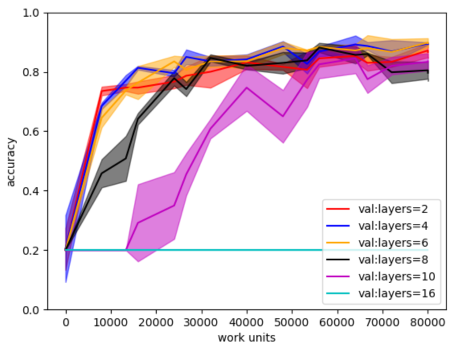
(a) Baseline



(b) Baseline with dropout $p = 0.5$

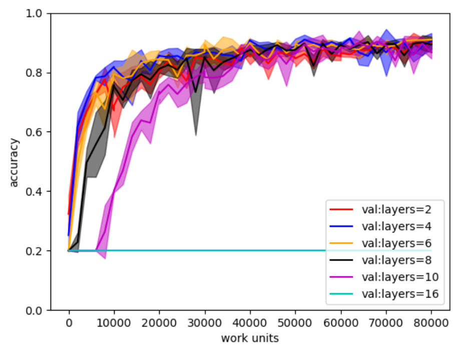


(c) MGDrop

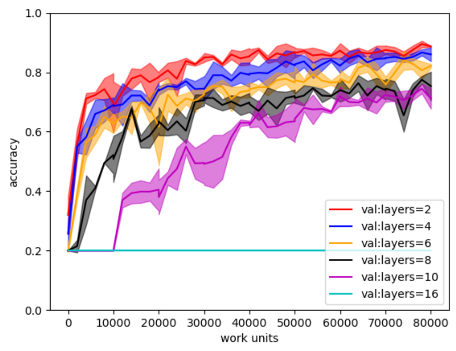


(d) SMGDrop

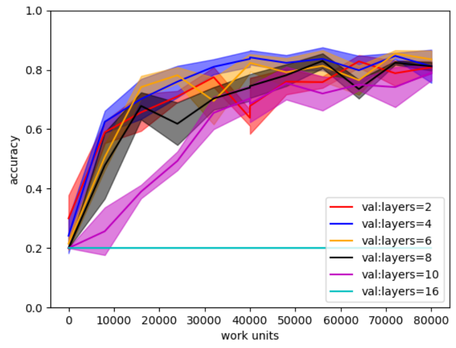
Figure 7.3: Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 128 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds.



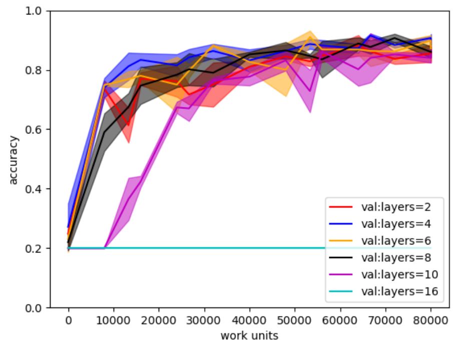
(a) Baseline



(b) Baseline with dropout $p = 0.5$

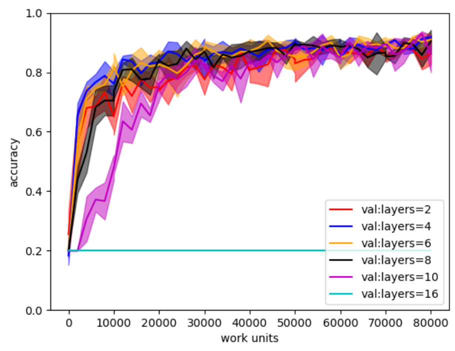


(c) MGDrop

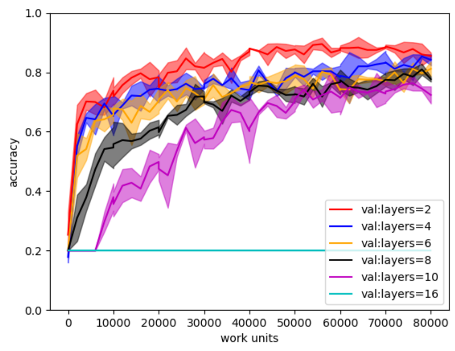


(d) SMGDrop

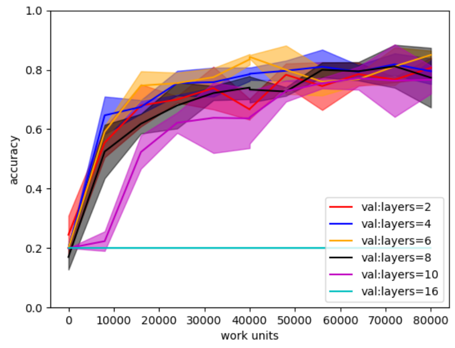
Figure 7.4: Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 512 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds.



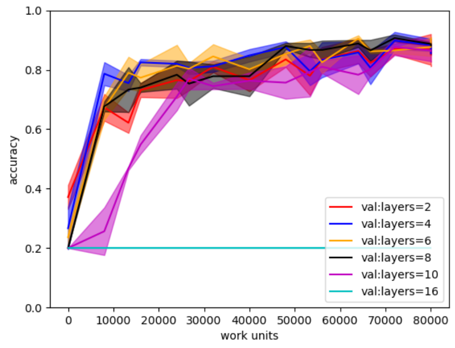
(a) Baseline



(b) Baseline with dropout $p = 0.5$

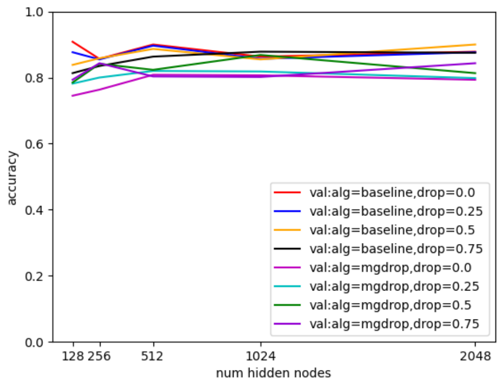


(c) MGDrop

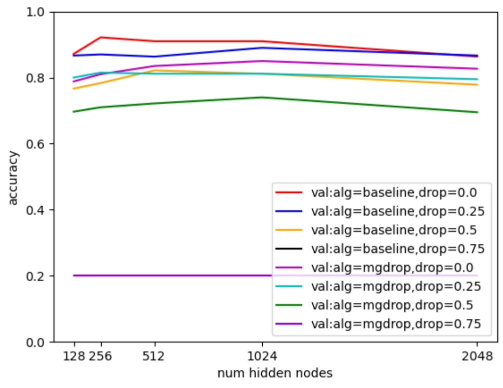


(d) SMGDrop

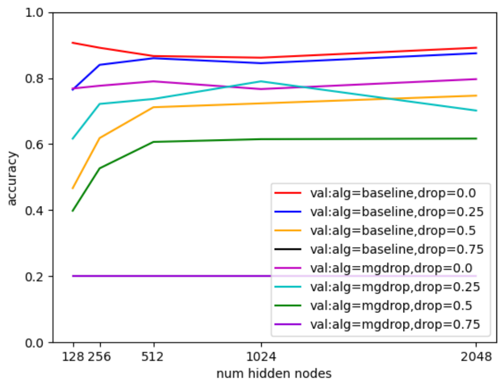
Figure 7.5: Performance of baseline without dropout, baseline with dropout of $p = 0.5$, MGDrop, and SMGDrop. Each plot shows the accuracy as a function of work units. The various curves represent networks with different numbers of layers. Each layer has 1024 nodes. The solid line for each curve represents the mean accuracy over three random seeds. The spread represents the standard deviation over the three random seeds.



(a) 2 layers

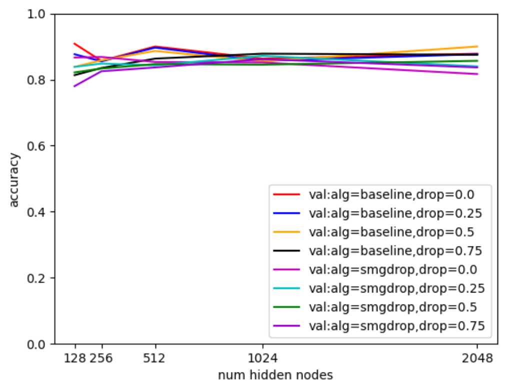


(b) 6 layers

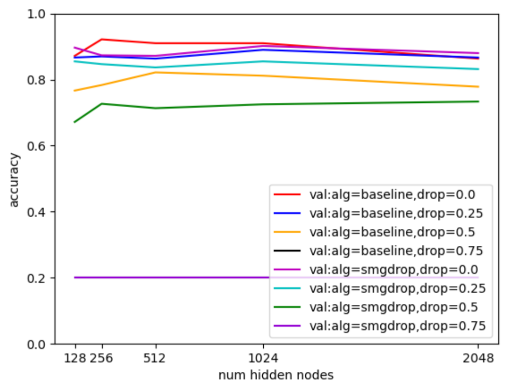


(c) 10 layers

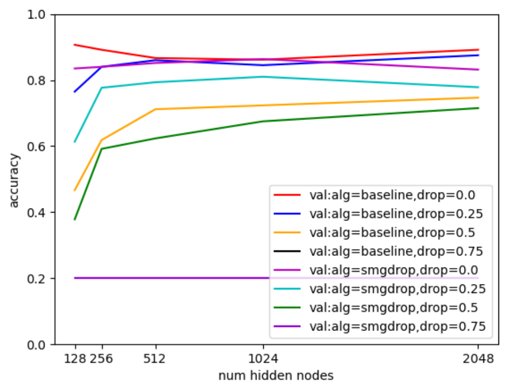
Figure 7.6: Performance of MGDrop for various dropout rates for 2, 6, and 10 layer networks. Baseline algorithms with the same dropout rates are shown for comparison.



(a) 2 layers

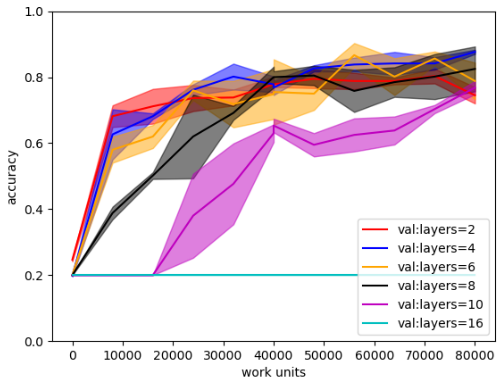


(b) 6 layers

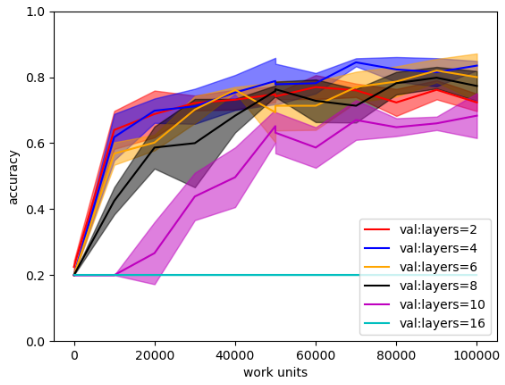


(c) 10 layers

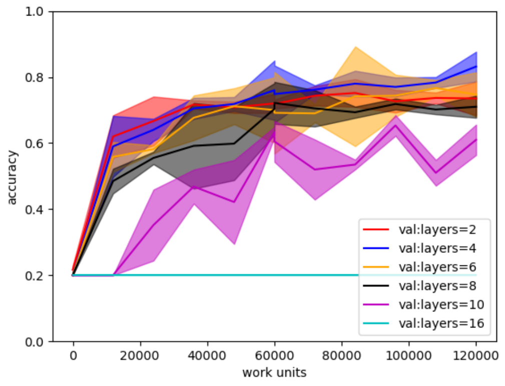
Figure 7.7: Performance of SMGDrop for various dropout rates for 2, 6, and 10 layer networks. Baseline algorithms with the same dropout rates are shown for comparison.



(a) $N_c = 1$

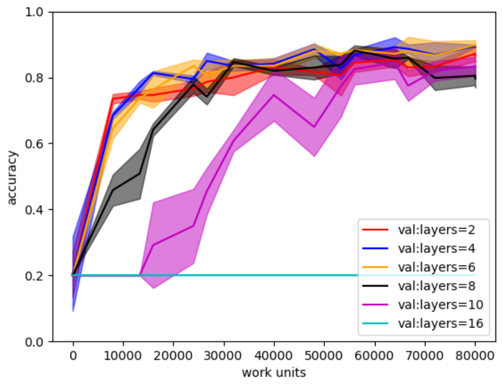


(b) $N_c = 2$

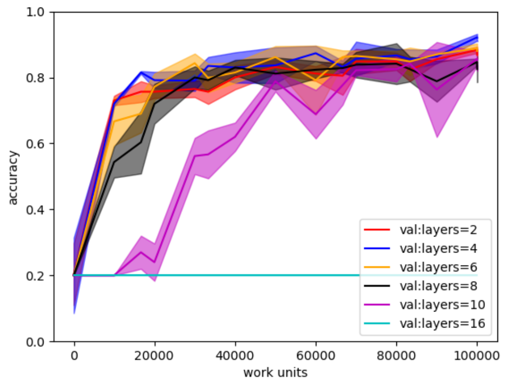


(c) $N_c = 3$

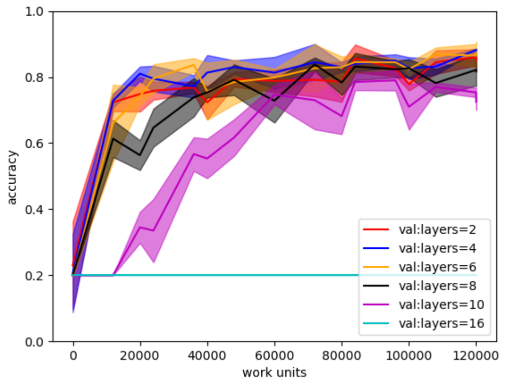
Figure 7.8: MGDrop Performance for different number of coarse optimization steps N_c. Each curve represents the performance for networks with different number of layers. Each layer has 128 nodes.



(a) $N_c = 1$

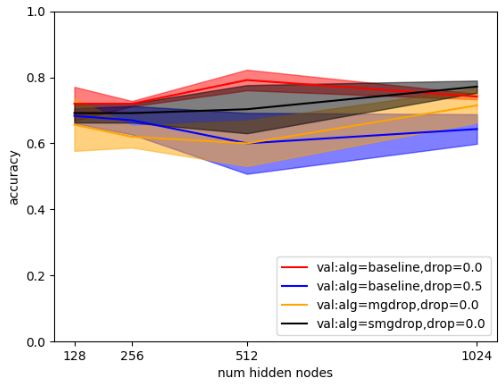


(b) $N_c = 2$

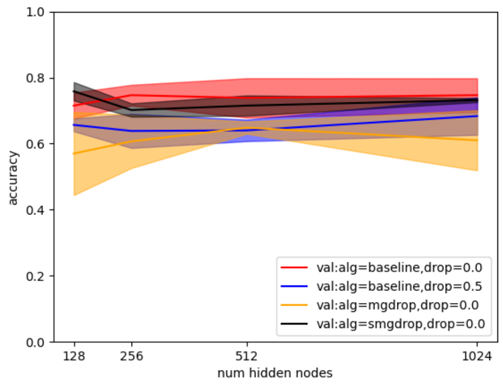


(c) $N_c = 3$

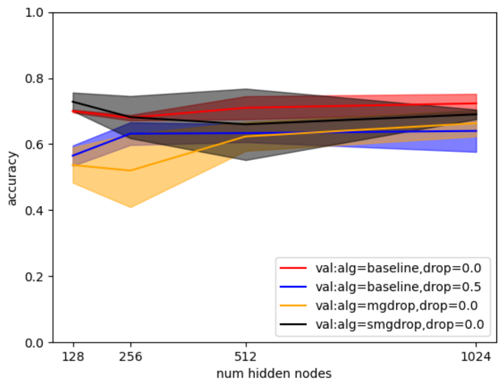
Figure 7.9: SMGDrop Performance for different number of coarse optimization steps N_c. Each curve represents the performance for networks with different number of layers. Each layer has 128 nodes.



(a) 10%

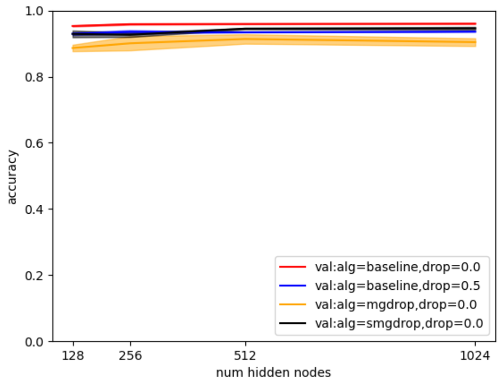


(b) 7.5%

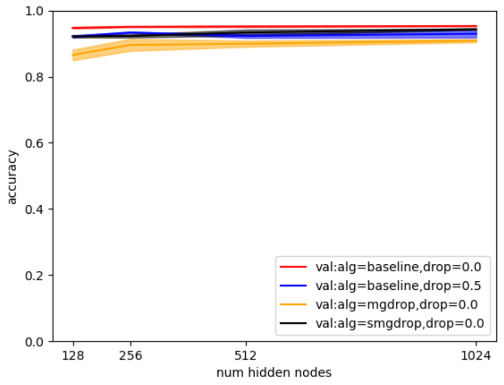


(c) 5%

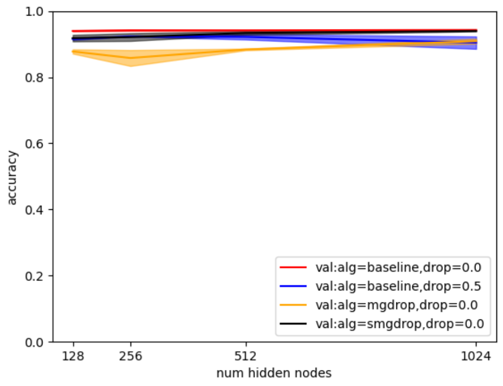
Figure 7.10: Performance of various algorithms for varying amounts of the Peaks training data.



(a) 10%



(b) 7.5%



(c) 5%

Figure 7.11: Performance of various algorithms for varying amounts of the MNIST training data.

Chapter 8

Conclusion

In this work we have shown that MG/OPT methods can offer improvements in NN training. We successfully utilize dropout as as a coarsening mechanism and show that coarsened NNs are good approximations of their fine counterparts. We analyzed the the performance of two novel algorithms MGDrop and SMGDrop. We showed that MGDrop performance was often comparable to baseline performance and in some cases was able to beat baseline algorithms. Further, by employing stochasticity, SMGDrop was able to improve over MGDrop's performance for a wide variety of network sizes and depths. Finally, we showed that MGDrop and SMGDrop tend to perform better than baseline when small amounts of data are used. This increase in performance can likely be attributed to dropout's ability to reduce overfitting, which would be more likely when only small amounts of data are available.

There are several avenues for future work. First, this study was limited to the Peaks dataset where overfitting is not present, and hence dropout is not as valuable. By broadening to new datasets where standard dropout shows improved performance, we may also see larger improvements in MGDrop and SMGDrop relative to baseline. Second, further experimentation with the tau correction term and its interaction with the NN loss function is warranted. Third, coarsening schemes besides dropout, for instance using Algebraic Multigrid, may provide additional training boosts.

References

[1] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[2] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis, 2013.

[3] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, 2021.

[4] Sunitha Basodi, Chunyan Ji, Haiping Zhang, and Yi Pan. Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3):196–207, 2020.

[5] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[6] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[7] Andriy Burkov. *The hundred-page machine learning book*, volume 1. Andriy Burkov Quebec City, QC, Canada, 2019.

[8] Eric C Cyr, Stefanie Günther, and Jacob B Schroder. Multilevel initialization for layer-parallel deep neural network training. *arXiv preprint arXiv:1912.08974*, 2019.

[9] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[10] Chen Dun, Cameron R Wolfe, Christopher M Jermaine, and Anastasios Kyrillidis. Resist: Layer-wise decomposition of resnets for distributed training. In *Uncertainty in Artificial Intelligence*, pages 610–620. PMLR, 2022.

[11] Adam Fadlalla and Chien-Hua Lin. An analysis of the applications of neural networks in finance. *Interfaces*, 31(4):112–122, 2001.

[12] Robert D Falgout, Stephanie Friedhoff, Tz V Kolev, Scott P MacLachlan, and Jacob B Schroder. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing*, 36(6):C635–C661, 2014.

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[15] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[16] Stefanie Gunther, Lars Ruthotto, Jacob B Schroder, Eric C Cyr, and Nicolas R Gauger. Layer-parallel training of deep residual neural networks. *SIAM Journal on Mathematics of Data Science*, 2(1):1–23, 2020.

[17] Guodong Guo and Na Zhang. A survey on deep learning based face recognition. *Computer vision and image understanding*, 189:102805, 2019.

[18] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse problems*, 34(1):014004, 2017.

[19] Mohammad Motamed. Approximation power of deep neural networks: an explanatory mathematical survey. *arXiv preprint arXiv:2207.09511*, 2022.

[20] Stephen G. Nash. A multigrid approach to discretized optimization problems. *Optimization Methods and Software*, 14(1-2):99–116, 2000.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and

Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[22] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[24] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681, 2019.

[25] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, 2000.

[26] Cyrill von Planta, Alena Kopanicáková, and Rolf Krause. Training of deep residual networks with stochastic mg/opt. *arXiv preprint arXiv:2108.04052*, 2021.

[27] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2022.

[28] Thorsten Wuest, Daniel Weimer, Christopher Irgens, and Klaus-Dieter Thoben. Machine learning in manufacturing: advantages, challenges, and applications. *Production & Manufacturing Research*, 4(1):23–45, 2016.