NRL Memorandum Report:

Gradually DropIn Layers to Train Very Deep Neural Networks: Theory and Implementation

Leslie N. Smith
Information Technology Division
Navy Center for Applied Research into Artificial Intelligence
U.S. Naval Research Laboratory
4555 Overlook Ave., SW., Washington, D.C. 20375

Emily M. Hand
University of Maryland
College Park, MD 20742

Timothy Doster
Optical Sciences Division
Applied Optics Branch
U.S. Naval Research Laboratory
4555 Overlook Ave., SW., Washington, D.C. 20375

David Aha
Information Technology Division
Navy Center for Applied Research into Artificial Intelligence
U.S. Naval Research Laboratory
4555 Overlook Ave., SW., Washington, D.C. 20375

March 10, 2016

Contents

List of Figures

List of Tables

Abstract

We introduce the concept of dynamically growing a neural network during training. In particular, an untrainable deep network starts as a trainable shallow network and newly added layers are slowly, organically added during training, thereby increasing the network's depth. This is accomplished by a new layer, which we call DropIn. The DropIn layer starts by passing the output from a previous layer (effectively skipping over the newly added layers), then increasingly including units from the new layers for both feedforward and backpropagation. We show that deep networks, which are untrainable with conventional methods, will converge with DropIn layers interspersed in the architecture. In addition, we demonstrate that DropIn provides regularization during training in an analogous way as dropout. Experiments are described with the MNIST dataset and various expanded LeNet architectures, CIFAR-10 dataset with its architecture expanded from 3 to 11 layers, and on the ImageNet dataset with the AlexNet architecture expanded to 13 layers and the VGG 16-layer architecture.

1 Introduction

Over the past couple of years, state-of-the-art results for image recognition [13, 26, 19], object detection [5], face recognition [27], speech recognition [7], machine translation [25], image caption generation [28], driverless car technology [11], and other applications [14] have required increasingly deeper neural networks.

Network depth refers to the number of layers in the architecture. It is well known that adding layers to neural networks makes them more expressive [15]. Each year, the Imagenet Challenge [18] is held in which teams are expected, given an image, to detect, localize, or recognize an object in the image. Deep convolutional neural networks (CNN) have dominated the competition since Krizhevsky *et al.*won in 2012 [13] and each year since the winner of the competition used a deeper network than the previous year's winner [18, 19, 26], e.g. the winner in 2012 used 8 layers while in 2014 the winner used 19 layers.

However, training a very deep network is a difficult and open research problem [4, 6, 22]. It is difficult to train very deep networks because the error norm during backpropagation can grow or vanish exponentially. In addition, very large training datasets are necessary when the network has millions or billions of weights.

Here we suggest a dynamic architecture that grows during the training process and allows for the training of very deep networks. We illustrate this with our DropIn layer, where new layers are skipped at the start of the training, as though they were not present. This allows the weights of the included layers to start converging. Over a number of iterations the DropIn layer increasingly includes activations from the inserted layers, which gradually trains the weights in theses added layers.

DropIn follows the philosophy embedded within curriculum learning [2]. With curriculum learning one starts with an easier problem and incrementally increases the difficulty. Here too, one starts training a shallow architecture and after convergence begins, DropIn incrementally modifies the architecture to slowly include units from the new layers.

In addition, DropIn can be used in a mode analogous to dropout [20] for the regularization of a deep neural network. Instead of setting random activations to zero as is done in dropout, DropIn set's the activations to that of the previous layer leading to a more robust trained network. This provides some of the benefits from regularization that dropout offers. In this report we demonstrate that the "noise" from mixing the activations from previous layers provides some regularization during training. In addition, both DropIn and dropout can be viewed as training a large collection of networks with varied architectures and extensive weight sharing.

The contributions of this report are:

1. Introduction of a dynamic architecture that grows or shrinks during the training.
2. The specifics of the DropIn layer for both enabling the training of very deep networks and for regularization during training. Also, a Resize layer is described to enable using DropIn even when the size of the input layers are different.
3. Examples of successfully training deep architectures that cannot be trained with conventional methods on MNIST, CIFAR-10, and ImageNet.
4. It is demonstrated that DropIn provides regularization during training.
5. A method to discover an optimal architecture for a given application and dataset.
6. Code is provide for reproducibility of this research.

2 Related Work

There has been a limited amount of work in recent years on how to train very deep networks. Methods for training very deep networks have centered on initialization of the network weights or developing new architectures and DropIn is in the latter category. We will now provide a brief review of the current state of initialization of network weights, new architecture for very deep networks, and regularization of networks. We will also mention how DropIn relates to these methods.

2.1 Initialization of Network Weights

Sutskever *et al.*[24] investigate the difficulty in training deep networks and conclude that both proper initialization and momentum are necessary. Glorot and Bengio [6] recommend an initialization method called "normalized initialization" to allow the training of deep networks. In this method weights are chosen from a distribution

$$W \sim U[-6/\sqrt{n_j + n_{j+1}}, 6/\sqrt{n_j + n_{j+1}}]$$

such that the activation variance and the backpropagation variance are maintained throughout the network. Using normalized initialization improved convergence. He *et al.*[8] recently improved upon the "normalized initialization" method by changing the distribution to take into account ReLU layers.

Hinton [9] proposed first training layer by layer in an unsupervised fashion so that a transformed version of the input could be realized. This method is able to capture the main variation of the input layer by layer. Finally supervised finetuning occurs, which has been setup to converge more easily by the pretraining phase. Erhan [4] would go on later to characterize the mathematics of the unsupervised pre-training and offer an explanation for its success.

Sussillo and Abbott [23] suggest an initialization scheme called "Random Walk Initialization" based on scaling the initial random matrices correctly. By multiplying the error gradient by a correctly scaled random matrix at each layer an unbiased random walk is formed. It can shown that the variance of the random walk grows only linearly with the depth of the network and thus the growing backpropagation error can be handled by increasing the size of the layers. This is one of only a few papers that show the results of experiments with networks consisting of hundreds of layers. The results in this paper indicated that just adding layers does not necessarily improve accuracy results, which we also found to be true.

2.2 Developing New Architecture

Raiko, *et al.*[16] introduce the concept of skip connections by adding a linear transformation to the usual non-linear transformation of the input to a unit. Skip-connections separate the linear and non-linear portions of the activations and allow the linear part to "skip" to higher layers. This is similar to DropIn in some ways, but the purpose of DropIn differs from that of skip connections, and DropIn does not need to learn any parameters. Furthermore, the skip connection experiments in their paper are with MNIST and CIFAR-10 datasets, which are shallow compared to the deep networks considered here.

Romero *et al.*[17] suggest training a thin, deep student network (called a *fitnet*) from a larger but shallower teacher network. The authors accomplish this by utilizing the output of the teacher's hidden layers as a hint for the student's hidden layers.

Srivastava *et al.*[21, 22] propose a new architecture that they named "Highway Networks" where the output of a layer's neuron contains a combination of the input and the output. Highway networks use carry gates inspired by long short-term memory (LSTM) recurrent neural networks (RNNs) to regulate how much of the input is carried to the next layer. The authors state that network depth is crucial for the recent success of neural networks and demonstrate that their structure permits training networks of hundreds of layers [21, 22] (up to 900 layers). These new parameters are learned along with the other parameters of the network. Zhang *et al.*[32] applied highway networks to LSTM recurrent neural networks. DropIn is a simpler approach than highway networks as it does not contain gates or their parameters, which need to be learned - which can result in a substantial savings of computation and storage.

Breuel [3] discusses a dynamic network that he describes as a biologically plausible "reconfigurable" network. In this network different units are more or less heavily weighted dynamically to produce different configurations and in this way a single network can perform multiple tasks. In other words, it is as though a network is composed of different classifiers, each implemented by a different configuration and each configuration is created by a control mechanism that sets an additional group of parameters. DropIn represents a different type of dynamic network that grows during training rather than reconfigures for each task.

2.3 Regularizing the Network

The well-known dropout [10, 20] method is an effective means to improve the training of deep neural networks. During training dropout randomly zeros a neuron's output activation with a probability p, called the *dropout ratio*, so that the network cannot rely on a particular configuration. This reduces overfitting to the training data and the

resulting network is more robust and better generalizes to unseen data. While dropout "samples from an exponential number of different 'thinned' networks" [20], DropIn samples from an exponential number of different thinner and shallower sub-networks. Like dropout, DropIn randomly changes the configuration so that the network cannot rely on a particular configuration.

The paper "Understanding Dropout" by Baldi and Sadowski [1] provides a theoretical basis for understanding dropout. The authors demonstrate that dropout is an approximation to averaging a large ensemble of networks, regulates the training, and prevents overfitting. A similar theoretical understanding and benefits can also apply to DropIn.

Wan *et al.*[29] suggest DropConnect, which randomly sets a subset of weights to zero (in contrast to randomly setting a subset of activations to zero, as is done by dropout). A recent paper by Wu *et al.*[30] suggests an interesting modification to dropout that they call split dropout. In each iteration, instead of zeroing activations or weights, they create two random, thinned sub-networks each iteration and train each separately.

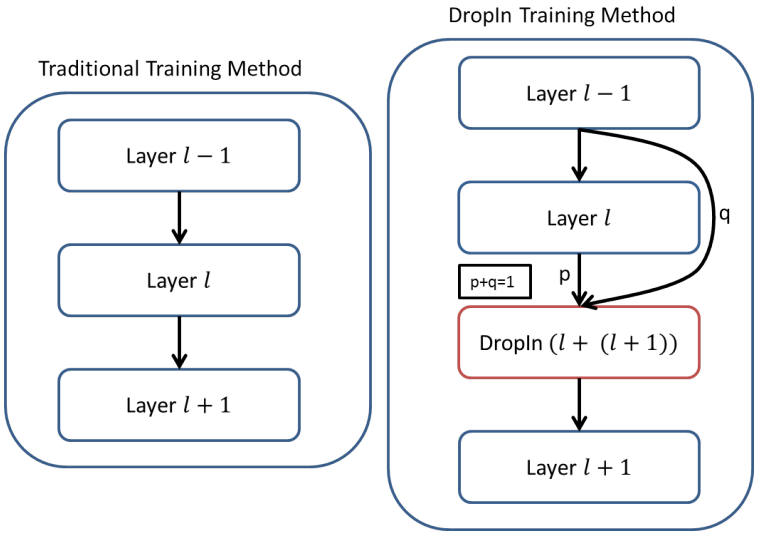


Figure 1: Diagram of traditional vs DropIn training method. The DropIn method sends activations from Layer $\ell - 1$ to Layer $\ell + 1$ (thus skipping Layer ℓ) with a ratio $q = 1 - p$ and from Layer ℓ to Layer $\ell + 1$ with a ratio p. (Best if viewed in color)

3 DropIn Method

In this section we provide a mathematical basis for DropIn as well as some implementation details.

3.1 Model Description

There are two modes of running DropIn: first to gradually include skipped layers, which we refer to as *gradual DropIn*, and second as a regularizer, which we named *regularizing DropIn*. Figure 1 provides a visual reference as to how the DropIn unit works.

Gradual DropIn initially passes on only the activations from the previous layer, effectively skipping the new layers. For each iteration number, τ, the ratio p is computed as $p = \tau/d$ for DropIn length d, which is the number of iterations over which $q = 1 - p$ reduces from 1 to 0. Then the number of activations copied from layer $\ell - 1$ drops as $q \times n = (1 - p) \times n$, where n is the total number of activations in the layer $\ell - 1$. The remaining activations are accepted from the new layer ℓ and backpropagation trains the weights of these newly added units.

For regularizing DropIn, the DropIn probability ratio p is set to a static value in $[0, 1]$. In this case, DropIn works analogously with dropout but instead of setting values to zero, they are set to the activations of a previous layer (e.g., $\ell - 1$). The choice of which activations come from which layer is done in an evolving random fashion each iteration.

We follow the notation in the dropout paper [20] to show this more formally. Namely, we start with a neural network composed of some number of layers, L, where $\ell \in [1, 2, ...L]$ is the layer index. Also, $\mathbf{y}^{(\ell)}$ represents the

vector of outputs from layer ℓ and is the input to the next layer $\ell + 1$. Let $\mathbf{x}$ be the data input to the first layer. In addition, $\mathbf{W}^{(\ell)}$ and $\mathbf{b}^{(\ell)}$ are the weights and biases at layer ℓ. To allow us to track the evolving nature of the network, we include the training iteration number, τ, and the layer's unit index number, $\lambda^{(\ell)}$.

The first equation for gradual DropIn (in the first mode where layers are slowly added) is a vector of zeros then ones, which is designated as $\mathbf{r}^{(\ell)}(\tau, \lambda)$ for each iteration:

$$\mathbf{r}^{(\ell)}(\tau, \lambda) = \begin{cases} 0 & \lambda < \Omega \\ 1 & \text{otherwise,} \end{cases} \tag{1}$$

where $\Omega = \min\{0, p \times n\}$. For regularizing DropIn, the equation for $\mathbf{r}^{(\ell)}(\tau, \lambda^{(\ell)})$ with a probability ratio p is:

$$\mathbf{r}^{(l)}(\tau, \cdot) \sim Bernoulli(p), \tag{2}$$

i.e., a 0-1 vector where each value is distributed as a Bernoulli random variable with probability p.

Once $\mathbf{r}$ is set, the remaining equations (dropping τ and $\lambda^{(\ell)}$ for simplicity) are the same for both modes – namely for layer $\ell + 1$:

$$\tilde{\mathbf{y}}^{(\ell)} = \mathbf{r}^{(\ell)} \times \mathbf{y}^{(\ell)} \tag{3}$$

$$z_i^{(\ell+1)} = \mathbf{w}_i^{(\ell+1)} \tilde{\mathbf{y}}^{(\ell)} + b_i^{(\ell+1)} \tag{4}$$

$$\mathbf{y}^{(\ell+1)} = f(z_i^{(\ell)}) + (1 - \mathbf{r}^{(\ell)}) \mathbf{y}^{(\hat{\ell})}, \tag{5}$$

where $\hat{\ell}$ is any layer less than layer $\ell + 1$.

These equations are similar to those for dropout, except instead of some of the outputs being zero, they are set to the values from the previous layer, $\mathbf{y}^{(\hat{\ell})}$. In Section 4.5 we look at whether DropIn offers some of the same regularization benefits as dropout.

3.2 Implementation

We implemented our code in Caffe [12] by creating a new layer called DropIn. The parameters for the DropIn layer (see Appendix B) include top, the top layer, $bottom$, a vector containing the bottom layer and the previous bottom layer, a $dropin\_ratio$, which is the ratio $q = 1.0 - p$ in Figure 1, and a $dropin\_length$, which is the number of iterations over which to q goes from 1.0 to 0.0 (i.e., it is d described in Section 3.1).

Since DropIn requires that the size of both the bottom layer and the previous bottom layer be the same, we also implemented a Resize layer (see Appendix A) to allow reshaping a layer to a user-specified size. The Resize layer modifies its input, which is $\mathbf{y}^{(\hat{\ell})}$, into a user-specified height, width, and number of channels/filters. The parameters are num, $channels$, $height$, $width$, and $back$, where num is the batch size, $channels$ is the number of filters, $height$ and $width$ are the height and width of the filters respectively, and $back$ is a boolean indicating if the resize is for back propagation. Since Caffe treats data as blobs, the actual method which performs the resizing is is added to the Blob class in Caffe. The bulk of the processing takes place in the actual Blob resize method. The Resize layer is implemented for CPU only because we found that using GPU for resize was actually slower than CPU because data needed to be copied back and forth between CPU memory and GPU memory, making the overall process slower. The Resize layer copies the input layer, calls the blob resize method, and returns the resized blob as the output. The Resize layer allows DropIn to work with any two layers, even when the sizes of $\mathbf{y}^{(\ell)}$ and $\mathbf{y}^{(\hat{\ell})}$ are different.

4 Experiments

The purpose of this section is to demonstrate the effectiveness of DropIn on several standard datasets but with deeper architectures. We trained DropIn networks on a variety of problems, in particular ones where the deep architecture was not trainable with standard methods. No attempt was made to optimize the architecture or hyper-parameters for higher accuracy because our main objective was to show that a deep architecture that will not converge without DropIn, will converge with it. However, the results in Sections 4.3 and 4.4 also demonstrate an increase in accuracy by using a deeper network for Imagenet.

LeNet	LeNet(2N) + DropIn
data (28×28)	
conv1_1-20(5×5)	conv1_1-20(5×5) conv1_2-20(3×3) dropin (1_1 + 1_2) conv1_3-20(3×3) dropin (1_2 + 1_3) $\vdots$ conv1_N-20(3×3) dropin (1_(N-1) + 1_N)
maxpool(2×2)	
conv2_1-50(5×5)	conv2_1-50(5×5) conv2_2-50(3×3) dropin (2_1 + 2_2) conv2_3-50(3×3) dropin (2_2 + 2_3) $\vdots$ conv2_N-50(3×3) dropin (2_(N-1) + 2_N)
maxpool(2×2)	
fc3-500	
fc4-10	
soft-max	

Table 1: Network architecture for LeNet and LeNet(2N)+ DropIn.

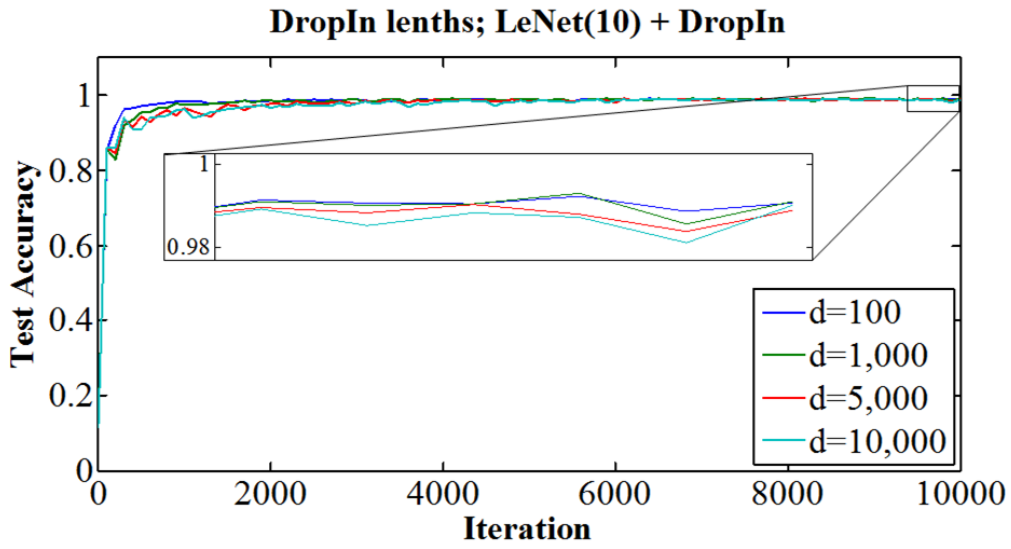
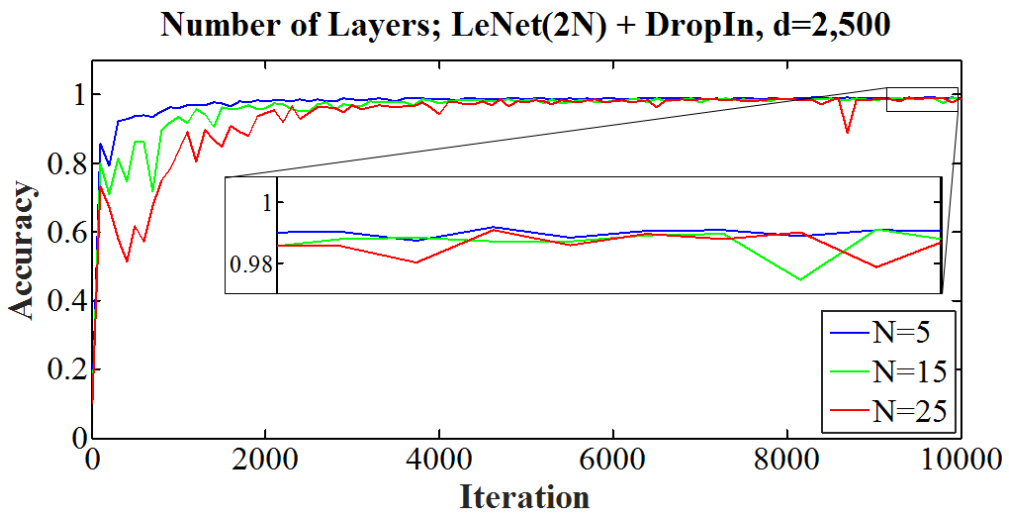


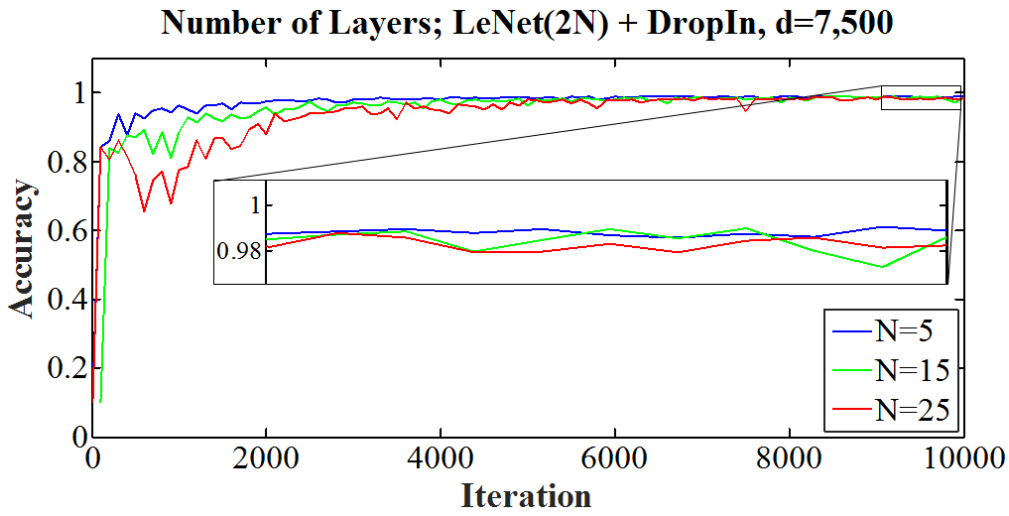
Figure 2: Classification accuracy while training LeNet(10) + DropIn architecture with MNIST data. Curves represent different DropIn lengths, d. (Best viewed in color)

In the subsections below, DropIn is used for training CNN architectures with the MNIST, and CIFAR-10 datasets, plus a modified AlexNet [13] with 13 layers and the VGG [19] architectures with the ImageNet dataset. These cases cover a range of data and architecture sizes. We found that in all cases DropIn permitted training an otherwise untrainable architecture.

All of the following experiments were run with Caffe (downloaded August 31, 2015) using CUDA 7.0 and Nvidia's



(a) DropIn length of 2,500



(b) DropIn length of 7,500

Figure 3: Classification accuracy while training LeNet(2N) + DropIn , for $N = 5, 15, 25, 35$ with MNIST data. Curves represent different network depths. (Best viewed in color)

CuDNN. The experiments described in this section were run on a 64 node cluster with 8 Nvidia Titan Black GPUs, 128 GB memory, dual Intel Xenon E5-2620 v2 CPUs, and 56 Gbps FDR InfiniBand (IB) per node.

The following subsections depict, in table form, the structure of several networks. We use the naming convention {layer type}{layer number}-{number of outputs}(filter size). For example, conv1_2-32(5×5) represents a convolutional layer numbered 1_2 with 32 outputs and filters sized 5×5. DropIn layers are denoted as dropin $(\ell + (\ell + 1))$, as depicted in Figure 1.

4.1 MNIST

This dataset consists of 70,000 grey-scale images with a resolution of 28x28[1]. Of these, 60,000 are for training and 10,000 are for testing. There are ten classes, each a different handwritten digit from zero to nine, with 7,000 images per class. The standard network architecture for the classification of MNIST, provided in the Caffe package, is the 4-layer

[1] http://yann.lecun.com/exdb/mnist/

LeNet consisting of 2 convolutional/max-pooling layers followed by 2 fully-connected layers (see the first column of Table 1 for details). Inspired by the work in [22], we increased the number of convolutional layers from two to 2N, which we denote as LeNet(2N). These added layers (as seen in the second column of Table 1, minus the DropIn layers shown in red) learned a 3×3 convolution filter but did not change the size of the outputs. We then added DropIn layers between each of the convolutional layers (as seen in the second column of Table 1) and called this network LeNet(2N) + DropIn.

We first looked at $N = 5$ and created LeNet(10) and LeNet(10) + DropIn architectures. LeNet(10) did not converge in the standard training time of 10,000 iterations given multiple realizations of the training process. However, utilizing DropIn units we were able to have LeNet(2N) + DropIn converge 10,000 iterations with the same hyper-parameters. In Figure 2 we show results for several different DropIn lengths for this network. These different lengths indicate the robustness of the DropIn length for simpler networks and that, in general, shorter DropIn lengths provide marginally better results. We note for this case that the added layers do not increase the overall accuracy of the network, as the MNIST data is quite simple compared with other classification tasks; the added layers do not provide any extra differentiation power.

We now look at how the number of layers affects the training with DropIn. In Figure 3 there are two different plots, one with DropIn length of 2,500 iterations and the other with DropIn length of 7,500 iterations. For each plot we present 4 different networks with 10, 30, 50, and 70 convolutional layers (equating to N=5, 15, 25, 35). For both DropIn lengths and all four network depths, the gradual DropIn method allowed the networks to converge. The deeper networks require a greater number of iterations to reach the same level of accuracy as the shallower networks, which is to be expected as they have a greater number of weights to train. We also see that networks converge more quickly with the shorter DropIn length, indicating that shorter DropIn lengths are desirable.

CIFAR-10	CIFAR-10(11 layers) + DropIn
data $(32 \times 32 \times 3)$	
conv1-32(5×5) maxpool(2×2) LRN	conv1_1-32(5×5) + LRN conv1_2-32(5×5) + LRN dropin (1_1 + 1_2)
conv2-32(5×5) maxpool(2×2) LRN	conv2_1-32(5×5) + LRN conv2_2-32(5×5) + LRN dropin (2_1 + 2_2)
	conv3_1-32(5×5) + LRN conv3_2-32(5×5) + LRN dropin (3_1 + 3_2)
	conv4_1-32(5×5) + LRN conv4_2-32(5×5) + LRN dropin (4_1 + 4_2)
	conv5_1-32(5×5) + LRN conv5_2-32(5×5) + LRN dropin (5_1 + 5_2)
conv3-64(3×3)	conv6_1-64(3×3)
maxpool(2×2)	
fc-10	
soft-max	
accuracy	

Table 2: CIFAR-10 11-layer architecture, including DropIn units. The layers follow the naming convention, {type}{layer number}{_sublayer number}-{number of outputs}{filter size}.

4.2 CIFAR-10

This dataset consists of 60,000 color images with a resolution of 32x32. Of these, 50,000 are for training and 10,000 are for testing. There are ten classes with 6,000 images per class.

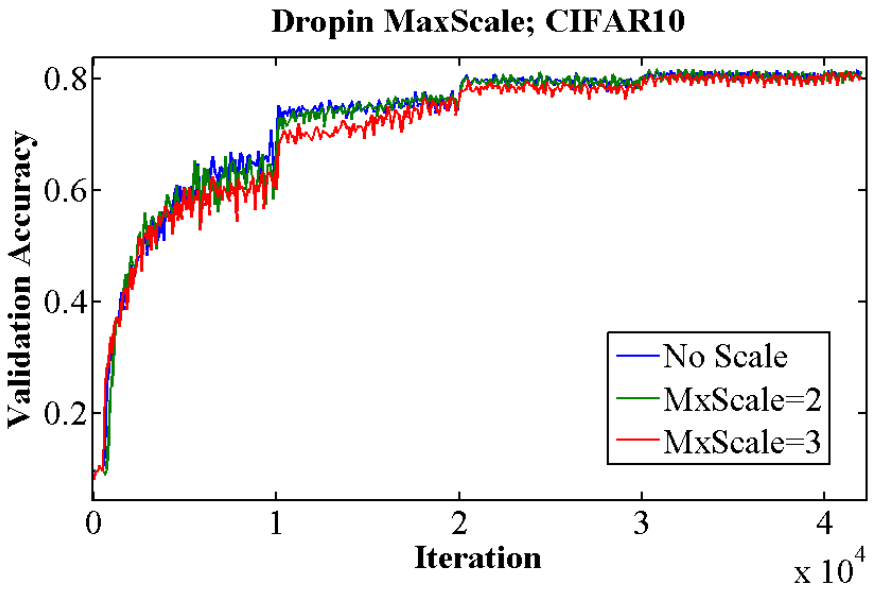


Figure 4: The curves show classification accuracies for no scaling and with scaling and different dropin_mxscale values. (Best if viewed in color)

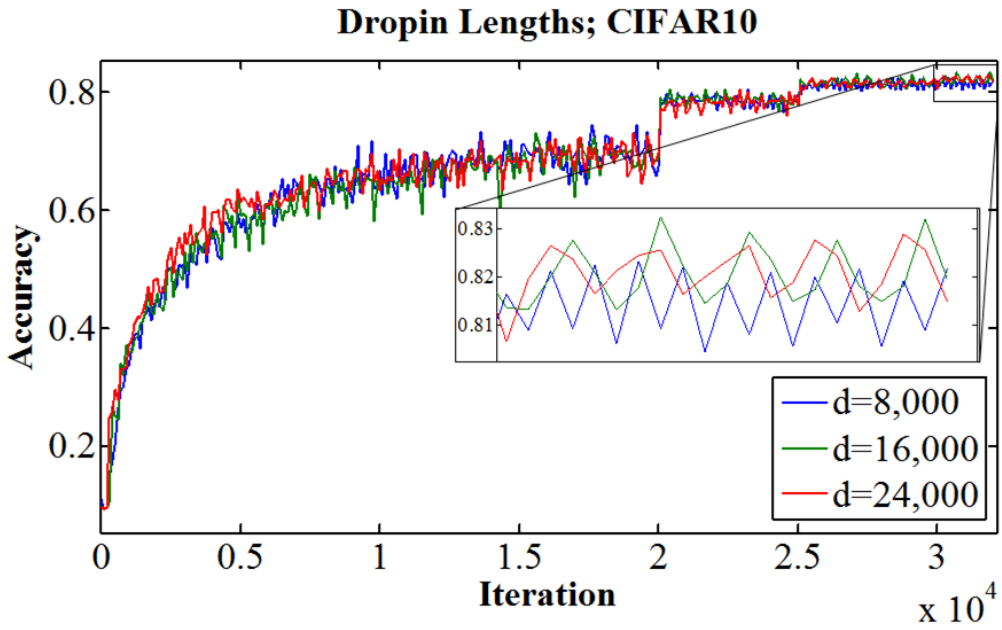


Figure 5: Test data classification accuracy while training the 11-layer CIFAR-10 architecture with DropIn. The curves show classification accuracies for different dropin_lengths, d. (Best viewed in color)

The Caffe [12] website provides a CIFAR-10 tutorial that we assumed to be a fairly standard architecture and hyper-parameter settings. These architecture and hyper-parameter files are available from the Caffe website [2] and were used here as a starting point.

The three convolutional layer architecture trains quickly and attains good accuracies. The convolutional layers were replicated to obtain an 11-layer model, which corresponds to the depth of one of the CIFAR-10 models in the experiments for highway networks [22]. The detailed architectures are compared in Table 2. As shown in the table, the

[2] http://caffe.berkeleyvision.org/gathered/examples/cifar10.html

Architecture	dropin_length	Accuracy (%)
3-layer net		81.4
11-layer net	8,000	81.7
11-layer net	16,000	82.3
11-layer net	24,000	82.3

Table 3: Final accuracy (average of last three values) results for the CIFAR-10 dataset on test data at the end of the training. Comparison of DropIn and dropin_lengths.

Architecture	dropin_length	Accuracy (%)
3 layer net		81.4
11 layer net	8,000	82.2
11 layer net	16,000	82.0
11 layer net	24,000	81.5

Table 4: Comparison of DropIn and dropin_length. The table shows accuracy results for the CIFAR-10 dataset on test data at the end of the training.

sizes of each of the layers entering the DropIn layer were kept the same for simplicity. For every convolutional layer, the weight initialization was Gaussian with standard deviation of 0.01 and the bias initialization was constant, set to 0. Each convolutional layer was followed by a rectified linear unit and local normalization. The parameters for every local normalization were local size = 3, alpha = 5×10^{-5}, beta = 0.75, and norm region = WITHIN CHANNEL. The length of the training, the learning rates, and schedule were modified to run over 32,000 iterations. This modification trained satisfactorily and provided a reasonable comparison. The full solver.prototxt file can be seen in Appendix C.

Numerous attempts to train this 11 layer network without the DropIn layers (as indicated in in Table 2) failed to start converging. On the other hand, similar attempts to train this network with the DropIn layers following every even numbered layer did successfully converge, which is the primary result of this study.

In addition, experiments were performed on the two DropIn parameters; dropin_length and dropin_mxscale. Scaling was included into the study because it is used with dropout but DropIn does not have the same justification for having scaling. It was found that scaling without limitation caused divergences to appear near when the probability ratios p and q were close to 0 or 1 because the scaling is proportional to the inverse of these ratios. Hence the scale factors were limited by a user specified factor, dropin_mxscale, as follows: $scale = min(dropin\_mxscale, 1/p)$. And Figure 4, which compares DropIn with no scaling to the cases with scaling but limited by dropin_mxscale, confirms that scaling is not necessary. In this Figure, the dropin_length was fixed at 18,000 iterations and dropin_mxscale was set to 1.0, 2.0, or 3.0.

Figure 5 shows the accuracy curves for $dropin\_length = 8,000, 16,000, 24,000$, and Table 4 compares the final accuracies. The final accuracies show a marginal improvement for longer lengths but for CIFAR-10 the results are relatively independent of the length value. Furthermore, the final accuracies from the 11-layer architecture are less than 1% better than the original 3-layer architecture, which implies that for the CIFAR-10 dataset, the deeper networker provides only marginal improvement.

4.3 ImageNet / AlexNet

ImageNet[3] [18] is a large image database based on the nouns in the WordNet hierarchy. This image database used for the ImageNet Large Scale Visual Recognition Challenge and is commonly used as a basis of comparison in the deep learning literature. The database contains 1.2 million training and 50,000 testing images covering 1,000 categories.

Fortunately, the Caffe website provides the architecture and hyper-parameter files for a slightly modified AlexNet[4]. We downloaded the architecture and hyper-parameter files from the website and we expanded the architecture from 8 layers to 13 layers by duplicating each of the convolutional layers, which is shown (minus the DropIn layers shown

[3] www.image-net.org/

[4] caffe.berkeleyvision.org/gathered/examples/imagenet.html

AlexNet	AlexNet (13 layers) + DropIn
data ($227 \times 227 \times 3$)	
conv1_1-96(11×11)	conv1_1-96(11×11) conv1_2-96(11×11) dropin (1_1 + 1_2)
maxpool(2×2) + LocalNorm	
conv2_1-256(5×5)	conv2_1-256(5×5) conv2_2-256(5×5) dropin (2_1 + 2_2)
maxpool(2×2) + LocalNorm	
conv3_1-384(3×3)	conv3_1-384(3×3) conv3_2-384(3×3) dropin (3_1 + 3_2)
conv4_1-384(3×3)	conv4_1-384(3×3) conv4_2-384(3×3) dropin (4_1 + 4_2)
conv5_1-256(3×3)	conv5_1-256(3×3) conv5_2-256(3×3) dropin (5_1 + 5_2)
maxpool(2×2)	
fc6-4096	
fc7-4096	
fc8-1000	
soft-max	

Table 5: Network architecture for AlexNet and modified version of AlexNet, AlexNet (13 layers) + DropIn . The layers follow the naming convention, {type}{layer number}{_sublayer number}-{number of outputs}{filter size}.

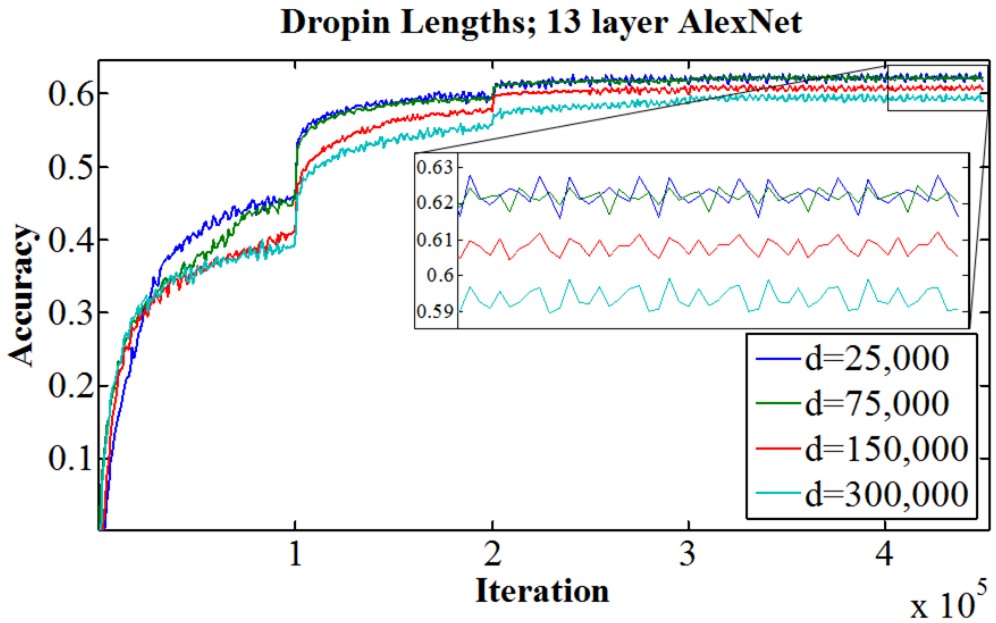


Figure 6: Comparison of various DropIn lengths, d. Validation data classification accuracy while training the AlexNet (13 layers) + DropIn architecture with ImageNet data. (Best viewed in color)

Architecture	dropin_length	Accuracy (%)
AlexNet		58.0
13 layers + DropIn	25,000	**62.2**
13 layers + DropIn	75,000	62.1
13 layers + DropIn	150,000	60.8
13 layers + DropIn	300,000	59.3

Table 6: Comparison of DropIn and dropin_lengths, d. The table shows final accuracy (average of last three values) results for the ImageNet dataset on validation data at the end of the training.

in red) in columns 1 and 2, respectively, of Table 5. The AlexNet (13 layers) + DropIn includes a DropIn layer between every duplicated layer used to create AlexNet (13 layers). Multiple attempts at training the AlexNet (13 layers) architecture in the conventional manner did not converge. In the training results reported in this section, all weights were initialized the same so as to avoid differences due to different random initializations. In the tests with the expanded architecture, the hyper-parameters were kept the same as provided by the Caffe website (even though our experiments with DropIn indicate that tuning them could improve the results, we left this for future work). In particular, the best learning rate and schedule is likely different so future work will include comparison with optimized hyper-parameters.

Experiments were run varying the DropIn hyper-parameter *dropin_length*. Table 6 shows final accuracy results after training for 450,000 iterations with a range of lengths. Figure 6 compares the accuracy during training of these experiments. In contrast to the results with CIFAR-10, the DropIn length makes a difference with ImageNet. We believe that this is because the deeper architecture increases the classification accuracy for larger datasets, hence the improvement with smaller DropIn lengths is more prominent.

From Figure 6 and Table 6, we can conclude that shorter lengths are better than the longer ones. If the length is less than the first scheduled drop in the learning rate at iteration 100,000, then the network is better trained. However, the difference between $dropin\_length = 75,000$ and 25,000 is negligible implying that lengths less than the first scheduled learning rate drop are equivalent.

4.4 ImageNet / VGG

VGGn, a set of networks created by the Visual Geometry Group [19], won second place in the image classification category of the 2014 ImageNet contest. These networks, trained on the same database as the Alexnet architecture discussed in Section 4.3, contained $n = 11, 13, 16,$ or 19 layers. In Table 7 we see the VGG16 (minus the DropIn layers shown in red) architecture alongside what we will refer to as VGG8 (not contained in the original paper). All convolutional layers have a stride and padding of 1 and maxpooling layers have a stride of 2. In their paper, the authors describe the difficulty of training these deep networks and utilized a weight transfer method to enable the network to converge during training.

In their paper the authors describe the difficulty of training these deep networks and utilized a weight transfer method:

> *The initialisation of the network weights is important, since bad initialisation can stall learning due to the instability of gradient in deep nets. To circumvent this problem, we began with training the configuration A (Table 1) [VGG13], shallow enough to be trained with random initialisation. Then, when training deeper architectures, we initialised the first four convolutional layers and the last three fullyconnected layers with the layers of net A (the intermediate layers were initialised randomly).*

While it is possible to train a deep neural network by first training a shallow network and using those weights to initialize the deeper network, we believe that in addition to being easier, training the full network with all the layers in place leads to a better trained network. This is supported by research on feature visualization, such as in Zeiler and Fergus [31], where they demonstrate that higher layers have more abstract representations. Training in place means that the learned representations will conform well to the representation at a given layer, while training a shallow network and initializing the weights of a deeper network might not. Our future work includes comparing using DropIn to initializing the weights from training a separate shallow network.

VGG8	VGG16	VGG16 + DropIn
data $(224 \times 224 \times 3)$		
conv1_1-64(3×3)	conv1_1-64(3×3) conv1_2-64(3×3)	conv1_1-64(3×3) conv1_2-64(3×3) dropin (1_1 + 1_2)
maxpool(2×2)		
conv2_1-128(3×3)	conv2_1-128(3×3) conv2_2-128(3×3)	conv2_1-128(3×3) conv2_2-128(3×3) dropin (2_1 + 2_2)
maxpool(2×2)		
conv3_1-256(3×3)	conv3_1-256(3×3) conv3_2-256(3×3) conv3_3-256(3×3)	conv3_1-256(3×3) conv3_2-256(3×3) dropin (3_1 + 3_2) conv3_3-256(3×3) dropin (3_2 + 3_3)
maxpool(2×2)		
conv4_1-512(3×3)	conv4_1-512(3×3) conv4_2-512(3×3) conv4_3-512(3×3)	conv4_1-512(3×3) conv4_2-512(3×3) dropin (4_1 + 4_2) conv4_3-512(3×3) dropin (4_2 + 4_3)
maxpool(2×2)		
conv5_1-512(3×3)	conv5_1-512(3×3) conv5_2-512(3×3) conv5_3-512(3×3)	conv5_1-512(3×3) conv5_2-512(3×3) dropin (5_1 + 5_2) conv5_3-512(3×3) dropin (5_2 + 5_3)
maxpool(2×2)		
fc6-4096		
fc7-4096		
fc8-1000		
soft-max		

Table 7: Network architectures for VGG8 and VGG16 + DropIn . The layers follow the naming convention, {type}{layer number}{_sublayer number}-{number of outputs}{filter size}. See the text for additional settings.

Instead of training smaller networks, we propose to use our gradual DropIn method. For our studies, we utilized the VGG16 prototxt file referenced on the Caffe website[5] and set up the solver file with the appropriate parameters from the authors' paper. Using traditional training methods, we were only able to train the VGG8 architecture; the VGG16 failed to begin converging for multiple realizations. Using VGG8 as a template, we augment VGG16 with DropIn layers to create VGG16 + DropIn (see Table 7). Due to the number of parameters in the VGG16 architecture we made use of the multigpu version of caffe by setting the batch size to 12 for 8 K40 GPUs - thus creating a combined batch size of 96, solver parameters and DropIn lengths were scaled to match this batch size. In Figure 7 we see the training accuracy results for DropIn of various lengths.

Based on the evidence presented in Section 4.3, we choose to test VGG16 with a DropIn length of 60,000. We found that other lengths (100,000, 150,000, and 200,000) began to converge as well but with limited time and resources, we chose to report only this length for this report. The results of training VGG16 + DropIn are shown in Figure 7. We see that with gradual DropIn the difficult to train VGG16 network does converge. Here we see the real power of the gradual DropIn method; without training an additional shallower network we are able to directly train VGG16, thus saving effort for the practitioner.

[5] https://gist.github.com/ksimonyan/211839e770f7b538e2d8#file-vgg_ilsvrc_16_layers_deploy-prototxt

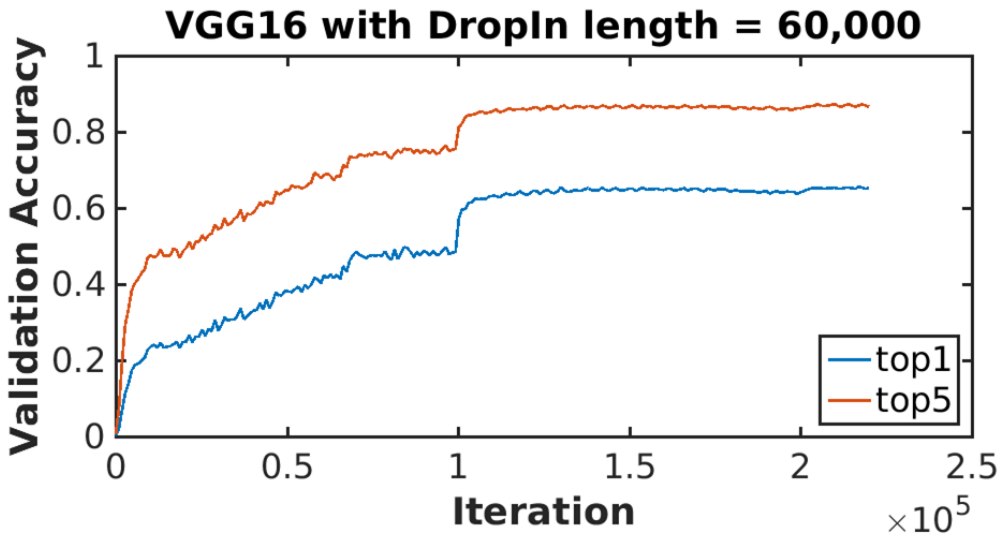


Figure 7: Validation data classification accuracy while training the VGG16 + DropIn architecture with ImageNet data.

4.5 Using DropIn for Regularization

The original AlexNet architecture uses dropout for regularization during training in both fully connected layers and it provides a substantial increase in the network's accuracy. AlexNet (with 8 layers) provides a means to test DropIn regularization. For this experiment, three cases were run as shown in Table 8. Case 1 is the original AlexNet.

Case	fc6	fc7
1	dropout	dropout
2	dropout	
3	dropout	DropIn

Table 8: The three regularization experiments shows layers with dropout or DropIn . The fully connected layers 6 and 7, are called *fc6* and *fc7*, respectively.

The results from this experiment are shown in Figure 8, where the blue curve is standard AlexNet, the green curve is without dropout in layer fc7, and the red curve is with DropIn in layer fc7. The dropin_ratio and dropout_ratio were 0.5 for all of these tests and all the hyper-parameters, such as learning rate, was kept the same.

This figure shows that removing dropout from fc7 causes visible degrading of the accuracy between iterations 150,000 and 200,000 (green curve). This kind of degradation does not happen with DropIn. Instead, the accuracy curve is similar to the curve with dropout (red versus blue curve) but with a small degradation in overall performance. We believe this degradation is because a DropIn network is more difficult to train than a dropout network. However, the final accuracy for the DropIn network is higher than from an architecture without dropout (red versus green curve). This experiment demonstrates that DropIn provides some regularization since the degradation found in the case without dropout is absent.

There are two implications from these results. First, using DropIn could be only partially utilizing the fc7 layer since half of those activations are replaced with fc6 activations. Second, the change to the architecture introduced by DropIn in fc7 might make it harder to train the network and that the hyper-parameters need to be tuned. Future plans include tuning the network to determine if the DropIn architecture can produce competitive accuracies to the original AlexNet.

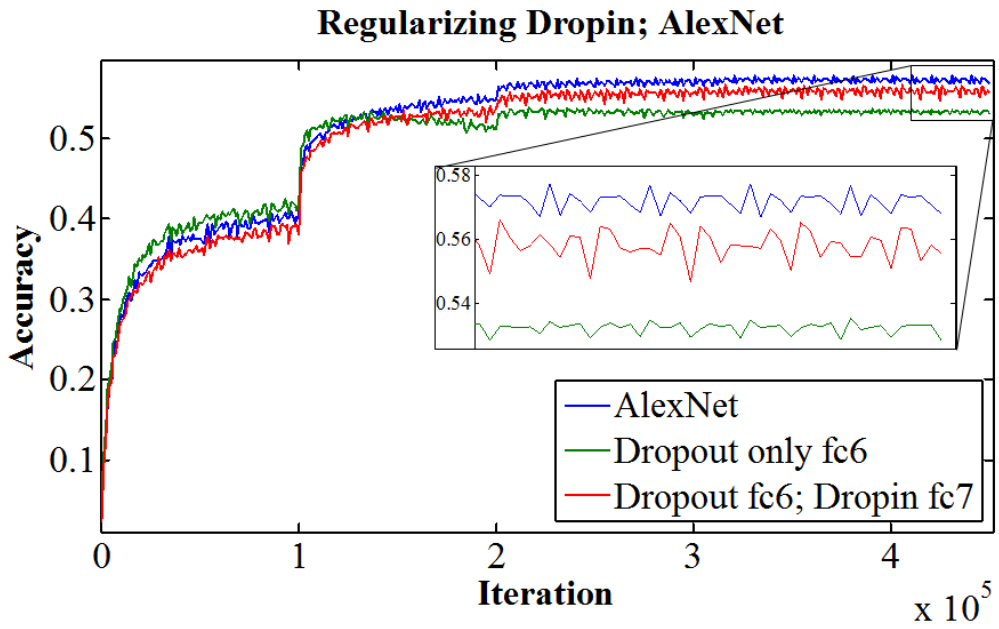


Figure 8: Test of DropIn regularization with AlexNet. Validation data classification accuracy while training AlexNet architecture with ImageNet data. See text for a discussion.

5 How to Easily Determine a Good Architecture

One of the challenges for deep learning practitioners is to determine good choices for the hyper-parameter values and the architecture for a given application and dataset. DropIn and dropout provide an easier way to test choices for the architecture than running a set of experiments with many different architectures.

DropIn and dropout can allow one to test a range of architecture depths and widths, respectively. Since adding layers does not necessarily increase accuracy, one can run with the gradual DropIn mode to see if there is little effect, such as in Figures 2 and 5, or visible effect, such as in Figure 6. Substantial improvement implies that there will be benefit from the additional depth.

Similarly, making a run where the dropout probability ratio varies from perhaps 0.9 to 0.1 (using a slightly modified dropout) provides guidance on the minimum number of neurons per layer. When decreasing the probability that neurons are retained (as shown in Figure 9 of Srivastava *et al.*[20]), the error typically has a range of the probability ratios where the error plateaus but at some threshold probability the error increases. By multiplying the number of neurons in a layer by this threshold probability, one can approximately determine the minimum number of neurons one must retain where there is negligible harm to the accuracy.

One can also start from fully trained weights and fine-tuning while letting the dropout ratio increase (thinner network) or letting the DropIn ratio increase (shallower network). If the effect on the accuracy is small and there is a need for computational savings, one can use the thinner and shallower network. If at a certain ratio for either dropout or DropIn there is an acceleration in the error, this ratio can be used to estimate the minimum depth and neurons per layer of the architecture for this dataset.

Furthermore, since DropIn and dropout allow for incremental changes to the architecture depth and width, one of our future research directions is to investigate automatic ways to learn an architecture during training.

6 Discussion & Conclusion

The major result of this report is that deeper architectures that cannot converge using standard training methods, become trainable by slowly adding in the new layers during the training. In addition, there are indications that DropIn layers help regularize the training of a network. We found in general that if the shallow network is trainable, then the deeper network, where additional layers are added by a DropIn layer, is also trainable. However, if the dataset is

relatively small, a deeper network might not provide much improvement, as shown with the CIFAR-10 results. With a large dataset like ImageNet, adding additional layers increases accuracy.

We have not yet explored training with different dropin_length values for different DropIn layers in one network. In addition, comparing DropIn to training by initializing the weights from training a separate shallow network has not yet been tested; these are planned for future work and will be reported elsewhere. Also, we plan to explore fine tuning the hyper-parameters for training the deep network with DropIn and to test DropIn within other architectures such as recurrent neural networks.

Future work also includes training networks with hundreds of layers using asynchronous DropIn, where layers are added starting at different iterations. In addition, we wish to test training where the entire very deep network is initially very thin (few parameters to train) and units are added to all the layers during the training. Furthermore, we plan to study if a methodology can be developed to learn from the data how to automatically optimize the architecture during training and thus learn to adapt to an application based on its data.

Acknowledgement

The authors express their appreciation to David Bonanno, Sambit Bhattacharya, and Michael Maynord for their suggestions and comments regarding this work. This work was supported by the US Naval Research Laboratory base program, Recursive Structure Learning.

Bibliography

[1] P. Baldi and P. J. Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, pages 2814–2822, 2013.

[2] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48. ACM, 2009.

[3] T. M. Breuel. Possible mechanisms for neural reconfigurability and their implications. *arXiv preprint arXiv:1508.02792*, 2015.

[4] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *International Conference on Artificial Intelligence and Statistics*, pages 153–160, 2009.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 580–587. IEEE, 2014.

[6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[7] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1764–1772, 2014.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[9] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[10] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[11] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, R. Cheng-Yue, F. Mujica, A. Coates, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *In Proceedings of the ACM International Conference on Multimedia*, pages 675–678, 2014.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.

[14] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[15] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2924–2932, 2014.

[16] T. Raiko, H. Valpola, and Y. LeCun. Deep learning made easier by linear transformations in perceptrons. In *International Conference on Artificial Intelligence and Statistics*, pages 924–932, 2012.

[17] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 2015.

[19] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[21] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[22] R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. *arXiv preprint arXiv:1507.06228*, 2015.

[23] D. Sussillo and L. Abbott. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*, 2015.

[24] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

[25] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[27] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708. IEEE, 2014.

[28] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[29] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[30] F. Wu, P. Hu, and D. Kong. Flip-rotate-pooling convolution and split dropout on convolution neural networks for image classification. *arXiv preprint arXiv:1507.08754*, 2015.

[31] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer, 2014.

[32] Y. Zhang, G. Chen, D. Yu, K. Yao, S. Khudanpur, and J. Glass. Highway long short-term memory rnns for distant speech recognition. *arXiv preprint arXiv:1510.08983*, 2015.

Appendix

A Resize

Passing the input from one layer to the next is complicated by the varying dimensions from one layer to the next. For DropIn , we have an interaction between three layers: L_{n-1}, L_n, and L_{n+1}. L_{n-1} has some number of channels, c_{n-1}, and some filter size $h_{n-1} \times w_{n-1}$. L_n is of size $c_n \times h_n \times w_n$. L_{n+1} expects input of size $c_n \times h_n \times w_n$, and so L_{n-1} must be resized to the size of L_n in order to properly pass the input of L_n to L_{n+1}.

We implemented a Resize layer in Caffe. It can be placed after any layer (convolution, ReLU, fully connected, etc.) and the new dimensions are specified there.

We tried several different methods for resizing L_{n-1}. Initially, to test our method, we used cropping and repeating values to resize L_{n-1}. We assume that $h_i = w_i$ with $i = 1, \ldots, m$ where m is the number of layers in the network. If $h_n < h_{n-1}$ (and therefore $w_n < w_{n-1}$), we simply crop each channel of L_{n-1} starting from the upper left and going to (h_n, w_n). If $h_n > h_{n-1}$, then we simply fill the rest of the filter with 0s. If $h_n = h_{n-1}$, then we keep all the filters the same. If $c_n < c_{n-1}$, then we take the first c_n channels and discard the rest. If $c_n > c_{n-1}$, then we repeat the last filter of $L_{n-1} c_n - c_{n-1}$ times.

The second iteration of the Resize layer uses OpenCV's Mat resize, and treats each filter as an image. It performs one of several interpolation methods for increasing or decreasing the size of a filter. To increase the size of the filter we use OpenCV's bicubic interpolation method and for decreasing the size of a filter, we use OpenCV's resampling using pixel area relation. If $c_n < c_{n-1}$, we do the same as before, and if $c_n > c_{n-1}$ we randomly choose $c_n - c_{n-1}$ of L_n's filters and add those to the end of L_{n-1}'s filters.

The following code is the implementation of the resize method on the CPU and GPU using C++ and CUDA.

resize.cpp

```
template <typename Dtype>
void ResizeLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    Blob<Dtype>* new_bottom= new Blob<Dtype>(bottom[0]->num(), bottom[0]->
        channels(), bottom[0]->height(), bottom[0]->width());
    new_bottom->CopyFrom((*bottom[0])); //copy only data
    new_bottom->Resize(top[0]->num(), top[0]->channels(), top[0]->height(),
        top[0]->width(), false);
    const Dtype* new_bottom_data = new_bottom->cpu_data();
    Dtype* top_data = top[0]->mutable_cpu_data();
    caffe_copy(top[0]->count(), new_bottom_data, top_data);
    free(new_bottom->mutable_cpu_data());
    new_bottom=NULL;
}

template <typename Dtype>
void ResizeLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom){
    if(propagate_down[0]){
        Blob<Dtype>* new_top = new Blob<Dtype>(top[0]->num(), top[0]->channels
            (), top[0]->height(), top[0]->width());
        new_top->CopyFrom((*top[0]), true); //Copy only diff
        new_top->Resize(bottom[0]->num(), bottom[0]->channels(), bottom[0]->
            height(), bottom[0]->width(), true);
        const Dtype* new_top_diff = new_top->cpu_diff();
        Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
        caffe_copy(bottom[0]->count(), new_top_diff, bottom_diff);
        free(new_top->mutable_cpu_diff());
```

```
        new_top = NULL;
    }
}
```

resize.cu

```
template <typename Dtype>
void Blob<Dtype>::Resize(const int num, const int channels, const int height,
    const int width, bool back) {
  CHECK_EQ(num, shape_[0]);
  CHECK_GE(channels, 0);
  CHECK_GE(height, 0);
  CHECK_GE(width, 0);
  cv::Mat cv_orig(shape_[2],shape_[3],CV_32FC1);
  cv::Mat cv_new;

  srand (time(NULL));

  if(num*channels*height*width == count_)
  {
    Reshape(num, channels, height, width);
    return;
  }
  int index=0;
  int oldIndex=0;
  Dtype* stuff;
  if(back)
    stuff = mutable_cpu_data();
  else
    stuff = mutable_cpu_diff();

  int old_count = shape_[0]*shape_[1]*shape_[2]*shape_[3];
  int new_count = num*channels*height*width;
  Dtype* stuff_copy = new Dtype[old_count];

  for(int i=0; i<old_count; i++)
  {
    stuff_copy[i] = stuff[i];
  }
  delete[] stuff;

  //Reset data or diff pointer memory
  if(back)
  {
    diff_.reset(new SyncedMemory(new_count * sizeof(Dtype)));
    stuff = mutable_cpu_diff();
  }
  else
  {
    data_.reset(new SyncedMemory(new_count * sizeof(Dtype)));
    stuff = mutable_cpu_data();
  }

  int INTER_METHOD=CV_INTER_CUBIC; //grow
```

```
48    if (width < shape_[3])
49      INTER_METHOD=CV_INTER_AREA; //shrink
50
51    int minChannels = channels;
52    if(shape_[1]<minChannels)
53      minChannels=shape_[1];
54    for(int n=0; n<shape_[0]; n++)
55    {
56      //copy the first channels over
57      for(int c=0; c<minChannels; c++)
58      {
59        for(int h=0; h<shape_[2]; h++)
60        {
61          for(int w=0; w<shape_[3]; w++)
62          {
63            index = ((n*shape_[1]+c)*shape_[2]+h)*shape_[3] + w;
64            cv_orig.at<float>(h,w) = stuff_copy[index];
65          }
66        }
67        cv::resize(cv_orig, cv_new, cvSize(width, height), INTER_METHOD);
68        for(int h=0; h<height; h++)
69        {
70          for(int w=0; w<width; w++)
71          {
72            index = ((n*channels+c)*height+h)*width+w;
73            stuff[index] = cv_new.at<float>(h,w);
74          }
75        }
76      }
77      if(shape_[1] < channels) //Randomly copy channels_ to fill the new channel
          size
78      {
79        for(int c=shape_[1]; c<channels; c++)
80        {
81          int randChan = rand() % shape_[1];
82          for(int h=0; h<height; h++)
83          {
84            for(int w=0; w<width; w++)
85            {
86              index = ((n*channels+c)*height+h)*width+w;
87              oldIndex = ((n*channels+randChan)*height+h)*width+w;
88              stuff[index] = stuff[oldIndex];
89            }
90          }
91        }
92      }
93    }
94
95    shape_[1] = channels;
96    shape_[2] = height;
97    shape_[3] = width;
98    count_ = num*channels*height*width;
99    capacity_ = count_;
```

```
100
101
102   delete[] stuff_copy;
103 }
```

B DropIn

The following code is the implementation of DropIn on the GPU using C++ and CUDA.

dropin.cu

```
1  #include <algorithm>
2  #include <limits>
3  #include <vector>
4  #include <iostream>
5
6  #include "caffe/common.hpp"
7  #include "caffe/layer.hpp"
8  #include "caffe/syncedmem.hpp"
9  #include "caffe/util/math_functions.hpp"
10 #include "caffe/vision_layers.hpp"
11
12 namespace caffe {
13
14 template <typename Dtype>
15 __global__ void DropinForward(const int n, const Dtype* in, const Dtype*
       prev_in,
16     const unsigned int* mask, const unsigned int threshold, const Dtype scale,
           const Dtype scale2,
17     Dtype* out) {
18   CUDA_KERNEL_LOOP(index, n) {
19 //    out[index] = in[index] * (mask[index] > threshold) + prev_in[index] * (
       mask[index] <= threshold);
20     out[index] = in[index] * (mask[index] > threshold) * scale
21                  + prev_in[index] * (mask[index] <= threshold) * scale2;
22   }
23 }
24
25 template <typename Dtype>
26 void DropinLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
27     const vector<Blob<Dtype>*>& top) {
28 //  std::cout<<"Forward 1\n";
29   const Dtype* bottom_data = bottom[0]->gpu_data();
30   const Dtype* prev_bottom_data = bottom[1]->gpu_data();//prev_bottom_blob->
       gpu_data();
31   const int count = bottom[0]->count();
32   Dtype* top_data = top[0]->mutable_gpu_data();
33
34   if (dropinGradually_) {
35     ++dropin_iter_;
36     threshold_ = std::max(0.0001, 1.0 - float(dropin_iter_) / length_);
37     uint_thres_ = UINT_MAX * threshold_;
38     if (maxScale2_ <= 1.0) {
39       scale_ = 1.0;
```

```
      scale2_ = 1.0;
    } else {
      scale_ = std::min(maxScale2_, float(1. / (1. - threshold_)) );
      scale2_ = std::min(maxScale2_, float(1.0 / threshold_) );
    }

    if (threshold_ > 0.99) {
      caffe_copy(count, prev_bottom_data, top_data);

    } else if (threshold_ > 0.01)  {
      int count2 = threshold_ * count;
      if (scale_ > 1.0) {
        caffe_gpu_scale(count2, scale2_, prev_bottom_data, top_data);
        caffe_gpu_scale(count-count2, scale_, bottom_data+count2, top_data+
            count2);
      } else {
        caffe_copy(count2, prev_bottom_data, top_data);
        if (top_data != bottom_data)  caffe_copy(count-count2, bottom_data+
            count2, top_data+count2);
        }

    } else {
      if (top_data != bottom_data) caffe_copy(count, bottom_data, top_data);
    }
  } else {
    unsigned int* mask =
        static_cast<unsigned int*>(rand_vec_.mutable_gpu_data());
    caffe_gpu_rng_uniform(count, mask);

    // NOLINT_NEXT_LINE(whitespace/operators)
    DropinForward<Dtype><<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(
        count, bottom_data, prev_bottom_data, mask, uint_thres_, scale_,
            scale2_, top_data);
    CUDA_POST_KERNEL_CHECK;
  }
}

template <typename Dtype>
__global__ void DropinBackward(const int n, const Dtype* in_diff,
    const unsigned int* mask, const unsigned int threshold, const Dtype scale,
        const Dtype scale2,
    Dtype* out_diff, Dtype* prev_out_diff) {
  CUDA_KERNEL_LOOP(index, n) {
     prev_out_diff[index] = in_diff[index] * (mask[index] <= threshold) *
         scale2;
     out_diff[index] = in_diff[index] * (mask[index] > threshold) * scale;
  }
}

template <typename Dtype>
void DropinLayer<Dtype>::Backward_gpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
```

```
88
89    if (propagate_down[0]) {
90      const Dtype* top_diff = top[0]->gpu_diff();
91      Dtype* bottom_diff = bottom[0]->mutable_gpu_diff();
92      Dtype* prev_bottom_diff = bottom[1]->mutable_gpu_diff();
93      const int count = bottom[0]->count();
94
95      if (dropinGradually_) {
96        if (threshold_ > 0.99) {
97         caffe_copy(count, top_diff, prev_bottom_diff);
98         CUDA_CHECK(cudaMemset(bottom_diff, 0.0, sizeof(Dtype)*count));
99
100       } else if (threshold_ > 0.01)  {
101         int count2 = threshold_ * count;
102         if (scale_ > 1.0) {
103           caffe_gpu_scale(count2, scale2_, top_diff, prev_bottom_diff);
104           caffe_gpu_scale(count-count2, scale_, top_diff+count2, bottom_diff+
                count2);
105         } else {
106           caffe_copy(count2, top_diff, prev_bottom_diff);
107           if (top_diff != bottom_diff) caffe_copy(count-count2, top_diff+
                count2, bottom_diff+count2);
108         }
109         CUDA_CHECK(cudaMemset(prev_bottom_diff+count2, 0.0, sizeof(Dtype)*(
              count-count2)));
110         CUDA_CHECK(cudaMemset(bottom_diff, 0.0, sizeof(Dtype)*count2));
111       } else {
112         if (top_diff != bottom_diff) caffe_copy(count, top_diff, bottom_diff);
113         CUDA_CHECK(cudaMemset(prev_bottom_diff, 0.0, sizeof(Dtype)*count));
114       }
115     } else {
116       const unsigned int* mask =
117           static_cast<const unsigned int*>(rand_vec_.gpu_data());
118
119       // NOLINT_NEXT_LINE(whitespace/operators)
120       DropinBackward<Dtype><<<CAFFE_GET_BLOCKS(count),
121         CAFFE_CUDA_NUM_THREADS>>>(
122           count, top_diff, mask, uint_thres_, scale_, scale2_, bottom_diff,
                prev_bottom_diff);
123       CUDA_POST_KERNEL_CHECK;
124     }
125   }
126 }
127
128 INSTANTIATE_LAYER_GPU_FUNCS(DropinLayer);
129
130 }  // namespace caffe
```

C Solver Parameters

The following is an example of the solver prototxt file used in the CIFAR experiment.

solver.prototxt

```
net: "conv11_dropin.prototxt"
test_iter: 100
test_interval: 100
momentum: 0.9
weight_decay: 0.004
# The learning rate policy
lr_policy: "fixed"
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.002
gamma: 0.25
# Display every 200 iterations
display: 100
# The maximum number of iterations
max_iter: 24000
# snapshot intermediate results
snapshot: 24000
snapshot_prefix: "snapshots/dropin24K"
# solver mode: CPU or GPU
solver_mode: GPU
```