Faster Convergence & Generalization in DNNs

Gaurav Singh
Department of Computer Science
University College London, UK
gaurav.singh.15@ucl.ac.uk

John Shawe-Taylor
Department of Computer Science
University College London, UK
j.shawe-taylor@ucl.ac.uk

Abstract

Deep neural networks have gained tremendous popularity in last few years. They have been applied for the task of classification in almost every domain. Despite the success, deep networks can be incredibly slow to train for even moderate sized models on sufficiently large datasets. Additionally, these networks require large amounts of data to be able to generalize. The importance of speeding up convergence, and generalization in deep networks can not be overstated. In this work, we develop an optimization algorithm based on generalized-optimal updates derived from minibatches that lead to faster convergence. Towards the end, we demonstrate on two benchmark datasets that the proposed method achieves two orders of magnitude speed up over traditional back-propagation, and is more robust to noise/over-fitting.

1 Introduction

Deep networks have gained immense popularity in the last few years. These networks were originally theorized in the 90's, but did not become immediately popular due to the lack computational power. Advancements in gpu technology over the years has led to these neural networks getting recognition. We have moved from using traditional multi-layer perceptrons (MLP) to more sophisticated networks like convolutional neural networks, recurrent neural networks, long shot-term memory networks etc.

These deep models are highly expressive and require large amounts of training data to avoid over-fitting. In addition, the training time required for most of these networks remains significant. Plus, there have been no theoretically sound ways to generalize these networks, except for using empirical methods like dropout and parameter weight regularization. Most theoretical analysis in the area begins by making assumptions on the structure of the network. Importance of the ability to train these networks in less time with smaller training data with generalization can not be overstated.

One possible direction for speeding up the training involves speeding up the computations involved in traditional back-propagation [11, 18]. It can improve the speed of computation, but does not reduce the number of epochs required for convergence. Such methods that improve on the computational efficiency of back-propagation do not lead to better generalization in the learned model. Also, the improvement in speed are generally not orders of magnitude higher over traditional optimizers.

Another direction to achieve speed up would be to improve upon traditional back-propagation i.e. using sgd or its variants. It has been shown that sgd consists of two phases: drift phase and diffusion phase [15]. It is well known that sgd is slow to converge and the reason for slow convergence is attributed to the diffusion phase, where learning rate needs to be lowered to avoid overshooting, leading to slow convergence. It is argued that the algorithm tries to compress the network during diffusion phase, and therefore does not focus on loss reduction.

One relevant work by Shawe-Taylor *et al.* [14] tried to approximate a neural network optimization process with a linear program. The linear program restricted the output nodes from moving in the wrong direction after update. It made a reasonable assumption that if the updates were small the

second order effects could be ignored even for a network with non-linear activations. But, the work did not make any efforts towards generalizing the updates. In this work, we aim to develop an approach that generates optimal and generalized updates that can lead to faster convergence with improved generalization of the model. The proposed method leads the model in the optimal direction to significantly decrease loss with each update.

Our contribution in this work can be summarized briefly as: 1) we develop an algorithm that computes generalized optimal updates that lead to faster convergence; 2) the proposed algorithm achieves two orders of magnitude speed up over traditional back-propagation on two benchmark datasets; 3) we bound the classification error of the network after k iterations; 4) we demonstrate that the learned network is more robust to adversarial noise and over-fitting.

2 Related Works

Deep neural networks started to gain popularity in late 90's [5, 9, 8]. These networks existed before in theory, but the lack of computational power required to train such network rendered them useless for most practical purposes. Advancement in gpu technology enabled orders of magnitude faster training for one particular type of deep networks, referred to as convolutional deep networks [8]. Consequently, these networks have been used to obtain state-of-the-art performances in image/text/video/audio classification tasks [7, 21, 16].

Further attempts to speed up computations of the convolution function in these networks have been made [11, 18]. One direction for making these deep convolution networks fast would involve speeding up of computations [11, 3] using techniques like fourier transforms. Another direction would be to approximate the computations exploiting the redundancies arising due to linear structure in these networks, instead of explicitly computing all floating point operations in the convolution layers [3].

Despite these attempts, the training process is still painfully slow, owing to the slow speed of convergence obtained using traditional back-propagation. The algorithm used often to train these networks is stochastic gradient descent. SGD has its merits, it is simple to implement; and acts like an implicit regularizer [22] by leading to solutions with smaller norm. But, it is slow to converge, as has been explained in Shwartz-Ziv *et al.* [15]. It shows that convergence through SGD has two phases: drift phase and diffusion phase. In the drift phase it explores the multidimensional space of solutions. When it begins converging, it arrives at the diffusion phase where it is extremely chaotic and the convergence rate slows to a crawl. It is argued that the network learns to compress during this phase.

Attempts to achieve faster convergence using a modification of the optimization algorithm have been made in the past, but with little success. Wilamowski *et al.* [19] used a modification of Levenberg-Marquardt algorithm (that relies on the second order derivatives) to get more optimal updates. One work that tried to fix the problems discussed above, and is nearest to this work, was done by Shawe-Taylor *et al.* [14]. It formulates neural network optimization as a linear program. It solves the LP to compute the most optimal update that leads to a movement in the correct direction for all output nodes simultaneously.

On the issue of generalization, there have been works to give theoretical bounds/measures. But, most such works make strong assumptions on the structure of the network. One such early work assumed that the network is linear and consists of 3 layers [12]. It considers an empirical bayes approach where some of the parameters are regarded as hyper-parameters. Other works made assumptions on the nature of regularizer [20] to obtain desired bounds on generalization.

3 Method

3.1 Generalisation from a mini-batch

The approach we adopt to choosing our update direction is motivated by an application of a learning theory bound that generalises from the mini-batch to the rest of the training and test examples. We choose a direction that improves the outputs on the majority of the mini-batch in terms of the hinge loss for each output label. Given that the mini-batch is a randomly chosen sample, and that the selected direction satisfies good bounds on generalisation, we infer that the update will not only improve the hinge loss of the mini-batch but also with high probability the hinge loss of the majority

of the rest of the training examples and a large fraction of the underlying distribution generating the test data.

Our analysis only applies if we ignore second order terms: $f^j(\mathbf{w}+\delta\mathbf{w},\mathbf{x}) \approx f^j(\mathbf{w},\mathbf{x}) + \langle \nabla f^j(\mathbf{w},\mathbf{x}), \delta\mathbf{w}\rangle$, where $f^j(\mathbf{w},\mathbf{x})$ is the j-th output of the network with weights assigned values $\mathbf{w}$ and input $\mathbf{x}$. Given the use of RELU activation functions, provided we are not at an inflexion point of the activation of any of the neurons for any of the inputs, there will be an open set around the current weight vector in which this equation will hold exactly. By controlling the norm of the update vector, we minimise the effect of the second order terms.

Hence, our task is to find an update direction $\delta\mathbf{w}$ that satisfies the following optimisation for a mini batch of size ℓ and a problem with K classes:

minimise $\frac{1}{2}\|\delta\mathbf{w}\|^2 + C\sum_{i=1}^{\ell}\sum_{j=1}^{K}\xi_{ij}$
Subject to: $y_{ij}\langle \delta\mathbf{w}_i, \nabla f^j(\mathbf{w},\mathbf{x}_i)\rangle \geq \epsilon - \xi_{ij}$
$\xi_{ij} \geq 0, i = 1,\ldots,\ell; j = 1,\ldots,K$

where ϵ is our target reduction in hinge loss for each example i and output class j. Note that y_{ij} indicates which class example i belongs to, controlling whether this class's output should be increased or reduced. Here we have introduced a parameter C to trade off the norm of the update vector with the overall amount by which points in the mini-batch fail to reduce their hinge loss by ϵ.

This optimisation corresponds exactly to a support vector machine with ℓK examples, but with the target output reduced from 1 to ϵ, and with the threshold fixed at 0. The dual of this optimisation is given by

Algorithm 1 Dual Update Calculation

max $\epsilon\sum_{ij}\alpha_{ij} - \frac{1}{2}\sum_{ijkl}\alpha_{ij}y_{ij}\alpha_{kl}y_{kl}\kappa((\mathbf{x}_i,j),(\mathbf{x}_k,l))$
subject to: $C \geq \alpha_{ij} \geq 0$,

where κ is the linear kernel taking inner products between the gradient vectors for the corresponding inputs/outputs, that is

$$\kappa((\mathbf{x}_i,j),(\mathbf{x}_k,l)) = \left\langle \nabla f^j(\mathbf{w},\mathbf{x}_i), \nabla f^l(\mathbf{w},\mathbf{x}_k)\right\rangle$$

This also means that we can apply standard margin based generalisation bounds to infer the expected reduction in hinge loss of samples outside the mini-batch, given the observed reduction in the mini-batch and the observed margin. We now introduce the relevant theoretical bounds.

Definition 1 (Rademacher complexity) *For a sample $S = \{\mathbf{x}_1,\ldots,\mathbf{x}_\ell\}$ generated by a distribution $\mathcal{D}$ on a set X and a real-valued function class $\mathcal{F}$ with domain X, the* empirical Rademacher complexity *of $\mathcal{F}$ is the random variable*

$$\hat{R}_\ell(\mathcal{F}) = \mathbb{E}_\sigma\left[\sup_{f\in\mathcal{F}}\left|\frac{2}{\ell}\sum_{i=1}^{\ell}\sigma_i f(\mathbf{x}_i)\right| \middle| \mathbf{x}_1,\ldots,\mathbf{x}_\ell\right], \tag{1}$$

where $\sigma = \{\sigma_1,\ldots,\sigma_\ell\}$ are independent uniform $\{\pm 1\}$-valued (Rademacher) random variables.

Theorem 1 *Fix $\delta \in (0,1)$ and let $\mathcal{F}$ be a class of functions mapping from Z to $[0,1]$. Let $(\mathbf{z}_i)_{i=1}^{\ell}$ be drawn independently according to a probability distribution $\mathcal{D}$. Then with probability at least $1-\delta$ over random draws of samples of size ℓ, every $f \in \mathcal{F}$ satisfies*

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}[f(\mathbf{z})] &\leq \hat{\mathbb{E}}[f(\mathbf{z})] + R_\ell(\mathcal{F}) + \sqrt{\frac{\ln(2/\delta)}{2\ell}} \\ &\leq \hat{\mathbb{E}}[f(\mathbf{z})] + \hat{R}_\ell(\mathcal{F}) + 3\sqrt{\frac{\ln(2/\delta)}{2\ell}}.\end{aligned}$$

Definition 2 *For a function $g: X \to \mathbb{R}$, we define its* margin *on an example $(\mathbf{x},y)$ to be $yg(\mathbf{x})$. The* functional margin *of a training set $S = \{(\mathbf{x}_1,y_1),\ldots,(\mathbf{x}_\ell,y_\ell)\}$, is defined to be*

$$m(S,g) = \min_{1\leq i\leq \ell} y_i g(\mathbf{x}_i). \tag{2}$$

Given a function g and a desired margin γ we denote by $\xi_i = \xi\left((\mathbf{x}_i, y_i), \gamma, g\right)$ the amount by which the function g fails to achieve margin γ for the example $(\mathbf{x}_i, y_i)$. This is also known as the example's slack variable

$$\xi_i = \left(\gamma - y_i g(\mathbf{x}_i)\right)_+, \tag{3}$$

where $(x)_+ = x$ if $x \geq 0$ and 0 otherwise.

Theorem 2 *[13] Fix $\gamma > 0$ and let $\mathcal{F}$ be the class of functions mapping from $Z = X \times Y$ to $\mathbb{R}$ given by $f(\mathbf{x}, y) = -yg(\mathbf{x})$, where g is a linear function in a kernel-defined feature space with norm at most B. Let*

$$S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\} \tag{4}$$

be drawn independently according to a probability distribution $\mathcal{D}$ and fix $\delta \in (0, 1)$. Then with probability at least $1 - \delta$ over samples of size ℓ we have

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}\left[\max(1, (\gamma - yg(\mathbf{x}))_+)\right] \quad \leq \quad & \frac{1}{\ell\gamma}\sum_{i=1}^{\ell}\xi_i + \frac{4B}{\ell\gamma}\sqrt{\mathrm{tr}(\mathbf{K})} \\ & +3\sqrt{\frac{\ln(2/\delta)}{2\ell}},\end{aligned}$$

where $\mathbf{K}$ is the kernel matrix for the training set and $\xi_i = \xi\left((\mathbf{x}_i, y_i), \gamma, g\right)$.

The theorem indicates conditions under which we can expect the reductions in hinge loss we have secured on the mini-batch will generalize to hinge loss reductions across the training and test sets. **Note**, the bounds depend on the norm of the svm (B) learned using data, and are therefore data dependant.

Lemma 1 *Let $\varepsilon(\gamma, \delta) = \mathbb{E}_{\mathcal{D}}\left[\max(1, (\gamma - yg(\mathbf{x}))_+)\right]$ (as defined in Theorem 2) and $H = \mathbb{E}_{\mathcal{D}}\left[\ell(f(x), y)\right]$ where H is the true hinge loss of the network. Then after $(i+1)_{th}$ iteration we have with probability at least $1 - \delta/k$:*

$$H_{i+1} \leq H_i - (\triangle_i - \varepsilon_i(\gamma_i, \delta/k)) + h_i \tag{5}$$

where $\triangle$ is the step size for the update (as defined in Algorithm 3.2) and h_i are the second order effects that we assume to be negligible.

Theorem 3 *Let n be the number of output nodes that have $\ell(.) > 0$ and*

$$S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\} \tag{6}$$

be drawn independently according to a probability distribution $\mathcal{B}$ such that $\mathcal{B}_{(x_i, y_i)} \propto n_i * \mathcal{D}_{(x_i, y_i)}$. Then applying Lemma 1 $k-$times and taking union over δ we have with probability at least $1 - \delta$:

$$H_k \leq H_0 - \sum_{i=1}^{k}(\triangle_i - \varepsilon_i(\gamma_i, \delta/k)) + \sum_{i=1}^{k} h_i \tag{7}$$

where H_0 is the initial hinge loss of the network, and H_k is the loss after k iterations.

Corollary 1 *Let $\hat{y} = \arg\max_j f^j(\mathbf{x}, \mathbf{w})$ and $e = \mathbb{1}(\hat{y} \neq y)$. Then $e = 1$ if and only if $\sum_j \ell(f^j(\mathbf{x}, \mathbf{w}), y^j) \geq 2$. We have with probability at least $1 - \delta$:*

$$E_k \leq \frac{1}{2} H_k \tag{8}$$

where $E_k = \mathbb{E}_{\mathcal{D}}\left[e\right]$ is the classification error of the network after k iterations.

3.2 Optimization

We use the updates derived in Section 3.1 to optimize the deep network. We compute gradient of each output node with respect to the parameters of the network given an instance x. We collect the gradients corresponding to all instances x in the minibatch. We feed these gradients as input to

a linear svm. The gradients of positive output nodes are labelled as $+1$ and gradients of negative output nodes are labelled as -1 before being passed to a binary linear svm. The svm returns the learned model (δw), which is then applied to the current parameters of the deep network after being multiplied by step size. This process is outlined in Algorithm 2. Note that it is not essential to consider every negative gradient corresponding to all classes for a given input x.

The value of $B\sqrt{tr(\mathbf{K})}/\ell$ (defined in Theorem 2) can be computed for each minibatch. If the generalization obtained is not satisfactory then the batch size can be doubled and svm can be more regularized to obtain a tighter bound. Once the sample size (ℓ) is sufficiently large it would ensure that each update derived from the minibatch reduces the error over both train and test set. We do not apply updates that have a loose bound as they would only diverge the optimization algorithm away from the solution and lead to slower convergence. The algorithm ensures that the loss is decreased by a certain amount over the both train and test sets.

Algorithm 2 TRAINING

```
function TRAINMINIBATCH(B, C, r)
    G ← {}
    for ∀(x, y) ∈ B do
        for o ∈ [1, 2 ··· |C| − 1, |C|] do
            g_o ← getGradient(x, y_o)          ▷ Obtain gradient of hinge loss on output node o
            if y_o = 1 then
                G ← G ∪ {(g_o, +1)}
            else
                G ← G ∪ {(g_o, −1)}
            end if
            δw ← trainSVM(G, r)
            return δw
        end for
    end for
end function
function TRAIN(D, batch_size)
    step_size = 0.1; r = 1.0
    for i ∈ [1, 2, ··· , num_iter] do
        B_s ← generateMinibatches(D, batch_size)
        δw ← trainMinibatch(B_s, C, r)
        bound_term = B√(tr(K))/ℓ
        if bound_term > threshold then
            batch_size ← 2 * batch_size
            r ← 0.1 ∗ r                        ▷ Regularizer for svm (lower value → more regularization)
        end if
        w ← w + step_size ∗ δw
    end for
    return w
end function
```

3.3 Cost comparison of updates

We currently take more run time than sgd for convergence with our crude implementation. We discuss the supposed causes that make our updates more expensive and describe technical/conceptual solutions to them.

- Sample-wise Gradients: We require gradients for each sample in the proposed algorithm. It is not efficient to obtain such gradients in existing libraries like tensorflow[1], since these libraries were designed to minimize a scalar loss. We need to obtain these gradients one at a time using a batch size of one. A library can be designed that could provide unaggregated gradients without the need to extract one at a time, therefore saving time due to concurrent computation and in-bulk copying of data to the gpu.

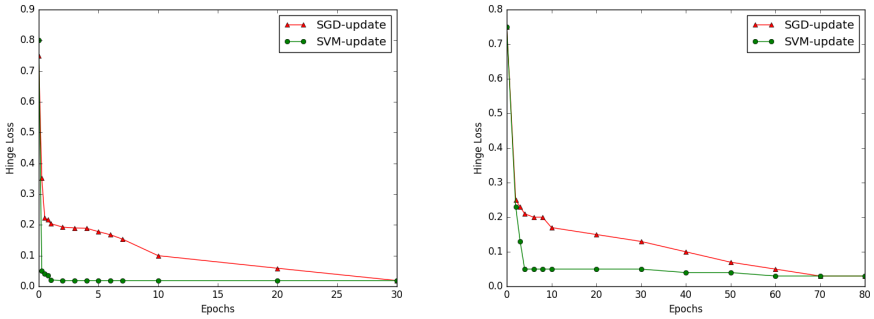

Figure 1: **Left**: mnist & **Right**: cifar. We can see the hinge loss over train set versus epochs. We observe that our svm-based updates converge much faster than the traditional back-propagation using sgd.

- Computing Kernel Gram Matrix: We can compute the kernel gram matrix required for solving the svm dual formulation in the gpu[4]. The kernel gram matrix corresponds to a minimatch, and has a reasonable size that can be computed in the gpu.
- Solving Svm: A lot of research has been done on efficiently solving an svm. Accelerated optimization algorithms are available [17]. Additionally, there have been some recent works to solve an svm using a gpu instead of the serial cpu implementation [2]. We only need to solve the svm for a minibatch of small size.

These technical improvements were beyond the scope of this work. Hence, we restrict the discussion to comparing different approaches in terms of epochs required for convergence instead of running time.

4 Experimentation

4.1 Dataset

We test our method on two bench-marking image datasets: CIFAR-10 [6] and MNIST [10].

- CIFAR-10 [6]: It consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The images have 3 channels, namely RGB, therefore have a depth of 3. It contains 10 classes e.g. cat, deer, dog, automobile etc., and classes are completely mutually exclusive.
- MNIST [10]: It consists of images of handwritten digits in binary. It has a training set of 60,000 examples, and a test set of 10,000 examples. The images are binary and each pixel value is either 0 or 1. The classes are again mutually exclusive from each other.

4.2 Experimental Settings

We used deep convolution networks for both mnist and cifar-10 classification task. For mnist, we constructed a network with two convolution layers of size 5x5 followed by a dense layer of size 30. The number of convolution filters used were 64 and 32 respectively. We experimented with different number of layers, and different layer sizes, but these gave the best results. For cifar, we constructed a slightly deeper network with 3 convolution layers of size 5x5 with depth 3 followed by a dense layer of size 30. The number of convolution filters used in each convolution layer were 120, 60 and 30 respectively. In both the networks, all the hidden layers are followed by a ReLU non-linearity and maxpooling layer of size 2x2 with strides of 2x2.

The hinge loss for a sample was computed as the average hinge loss over all output nodes. We set aside a separate validation set for parameter tuning. The parameters pertaining to the best performance on validation set were saved for prediction. We had a test set that was kept unseen during the entire training and development process. We experimented with different types of activations apart from ReLU, but did not observe a significant difference in the performance.

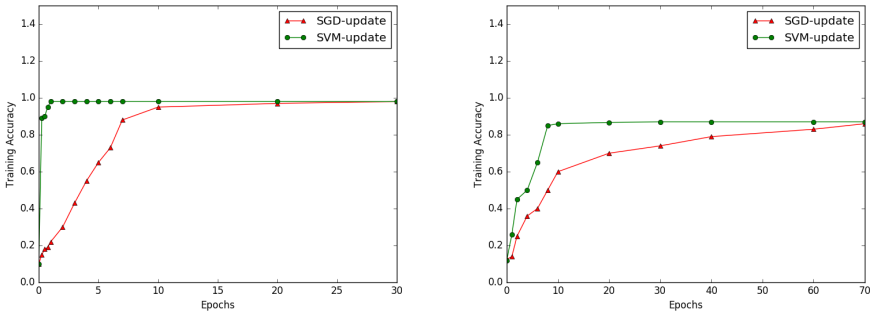

Figure 2: **Left**: mnist & **Right**: cifar. We can see the training accuracy over the entire train set versus epochs. We can see that our svm based updates converge much faster than the traditional back-propagation using sgd.

Dataset	svm-update	sgd-update
mnist	0.989	0.982
cifar	0.878	0.867

Table 1: We can see that the results obtained on the test set are only slightly different. Both the methods were tuned over a validation set of same size, and the best performing parameters were chosen to evaluate over test set. **Note** that we are not using the state-of-the-art networks for the tasks, but standard deep convolutional networks.

4.3 Results

We showed in Sec 3.1 that we can compute generalized updates from the minibatch that lead to decrease in the loss over entire training set, as well as test set. We can see in Figure 1 that the loss decreases much faster using our algorithm as compared to sgd. We observed a speed up of nearly two order of magnitude in terms of epochs required for convergence in mnist dataset, and one order of magnitude in convergence over cifar dataset. We can see in Figure 2 that the same trend follows for training accuracy.

We require one epoch for convergence over mnist as compared to thirty epochs required by sgd optimizer. Similarly, we require five epochs for convergence over cifar as opposed to seventy epochs required by sgd optimizer. These are significant speed ups, and even though our updates are more expensive, they can still give a significant overall speed up over traditional back-propagation.

We do not observe significant differences in the prediction accuracy of both sgd and our approach over test set. Both methods lead to similar accuracy over the test set as can be seen in Table 1. We used the same validation set to tune the parameters for both the methods. But, as can be seen in Figure 3, we do not require early stopping with our algorithm, which is another advantage of our method. As our updates are generalized we do not need to do nested validation in order to obtain the best results. We only show the results over mnist due to space constraints, but results on cifar were similar.

4.4 Robustness

One very important aspect of a learned network in deep learning is robustness to noise. We observed that both the update algorithms i.e. traditional back propagation and our svm-based updates perform well under random noise and do not show divergent behaviour. But, we observed that the proposed svm-update algorithm made the learned network more robust to additive adversarial noise. Robustness to adversarial noise is important to protect the network during an attack. We can see in Table 2 that the norm of the noise required to force the network into misclassification is much higher in our case compared to sgd. It shows that our generalized updates lead to a more robust network.

Norm	sgd-based	svm-based
frobenius	5.12	7.32
infinity	5.90	8.11
nuclear	7.29	8.11
1-norm	6.23	7.59

Table 2: We give details of the additive adversarial noise learned for mnist using traditional back-propagation and svm-based updates. Additive adversarial noise is the minimum amount of noise to be added to images such that the network misclassifies them.

4.5 Data-dependence

We experiment with fitting random labels to a network using sgd and our proposed svm-updates. It is well known that deep neural networks have high finite sample expressivity [22] and can memorize even randomly assigned labels [22]. We show in Section 3.1 that our updates are generalized and the bounds depend on the data. Despite that we decided to conduct experiments to observe the behaviour of our algorithm when faced with randomly assigned labels. We hoped that the bounds generated for a minibatch in such a scenario would grow too large and indicate that the data can not be used for learning the network. We indeed observed that sgd was able to overfit the randomly assigned labels, but our bound grew larger. We show empirical evidence to that effect in Figure 3. We can see that bound grows larger when the loss is only decreasing over the minibatch and not over the entire train set.

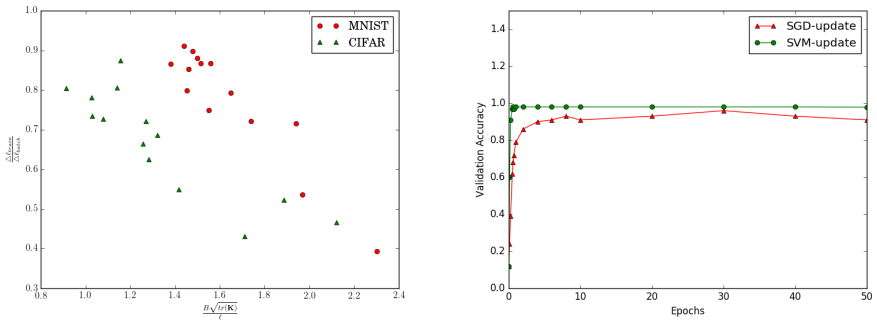

Figure 3: **Left**: We see that the ratio of decrease in loss over train set and mini-batch decreases with increase in the bound (see Theorem 2), implying that the updates become less generalized. **Right**: We observe that the validation accuracy initially increases and then stabilizes for mnist using our algorithm, as opposed to sgd.

5 Conclusion

We develop an algorithm based on generalized updates that can lead to faster convergence in deep networks. We prove that the updates are guaranteed to decrease the loss over the train and test set under certain conditions. Specifically, we achieve two orders of magnitude speed up compared to back-propagation over the same network on one of the dataset and one order of magnitude speed up over the other. We also show that the learned network is more robust to adversarial noise and over-fitting. We provide theoretical bound on the error of the network after k updates. In the future, we hope to extend the idea towards speeding up convergence in RNNs that are far more slower to train compared to deep CNNs.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale

machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] A. Cotter, N. Srebro, and J. Keshet. A gpu-tailored approach for training kernelized svms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 805–813. ACM, 2011.

[3] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[4] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137. ACM, 2004.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[8] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Y. Lecun, C. Cortes, and B. Christopher. Jc,the mnist database of handwritten digits., 2016.

[11] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[12] S. Nakajima and S. Watanabe. Generalization error of linear neural networks in an empirical bayes approach. In *IJCAI*, pages 804–810, 2005.

[13] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.

[14] J. S. Shawe-Taylor and D. A. Cohen. Linear programming algorithm for neural networks. *Neural Networks*, 3(5):575 – 582, 1990.

[15] R. Shwartz-Ziv and N. Tishby. Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*, 2017.

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

[17] T. Sun, H. Wang, Y. Shen, and J. Wu. Accelerating support vector machine learning with gpu-based mapreduce. In *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, pages 876–881. IEEE, 2015.

[18] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[19] B. M. Wilamowski, S. Iplikci, O. Kaynak, and M. O. Efe. An algorithm for fast convergence in training neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, pages 1778–1782. IEEE, 2001.

[20] P. Xie, Y. Deng, and E. Xing. On the generalization error bounds of neural networks under diversity-inducing mutual angular regularization. *arXiv preprint arXiv:1511.07110*, 2015.

[21] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[22] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.