Training Neural Networks Using Predictor-Corrector Gradient Descent: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4–7, 2018, Proceedings, Pa...

Chapter · October 2018

DOI: 10.1007/978-3-030-01424-7_7

CITATIONS
0

READS
119

2 authors:

Amy Nesky
Google Inc.
7 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Q. F. Stout
University of Michigan
251 PUBLICATIONS **5,241** CITATIONS
SEE PROFILE

Training Neural Networks Using Predictor-Corrector Gradient Descent

Amy Nesky and Quentin F. Stout

University of Michigan, Computer Science and Engineering, Ann Arbor, MI 48109
{anesky,qstout}@umich.edu

Abstract. We improve the training time of deep feedforward neural networks using a modified version of gradient descent we call Predictor-Corrector Gradient Descent (PCGD). PCGD uses predictor-corrector inspired techniques to enhance gradient descent. This method uses a sparse history of network parameter values to make periodic predictions of future parameter values in an effort to skip unnecessary training iterations. This method can cut the number of training epochs needed for a network to reach a particular testing accuracy by nearly one half when compared to stochastic gradient descent (SGD). PCGD can also outperform, with some trade-offs, Nesterov's Accelerated Gradient (NAG).

Keywords: Neural Networks · Accelerated Gradient Methods

1 Introduction

The immense expressional power of artificial neural networks has advanced machine learning and data science a great deal. Large networks can achieve unprecedented accuracy in intricate learning problems, yet their size consumes significant computational resources and, consequently, time [13]. Advances in compute power allow neural networks with millions of parameters to be trained on enormous, complex data sets, and the use of GPUs has decreased training time drastically, but new techniques for reducing network training time must arise for deep learning to progress.

In this work, we propose a new training technique called Predictor-Corrector Gradient Descent (PCGD) that reduces the number of iterations required to learn. In PCGD we monitor the trend of the parameters as the network learns with gradient descent, and periodically adjust each parameter by inferring future values from the trend. A number of standard gradient descent iterations between predictions act to refine the predicted approximations. This alternating process works in much the same way that predictor-corrector methods for solving ordinary differential equations work. We will show that incorporating prediction into the training process of networks makes learning significantly more efficient.

The human brain already utilizes predictions. Predictions are crucial to survival because they allow us to respond more appropriately to our surroundings and they improve reaction time. Perception is also impacted by brain predictions: our perceptions are a combination of expectations and sensory information [7,

14]. Thus, if we wish to improve artificial neural network efficiency, integrating prediction into training is a natural modification.[1]

2 Related Work

There is a plethora of work that supplements standard gradient descent in hopes of improving neural network training. Gradient noise and stale gradients have been successful adaptations to gradient descent [8, 15]. Adapative Gradient techniques give frequently occurring features low learning rates and infrequent features high learning rates; these methods use the information theoretic idea that infrequent features carry more information about the data distribution [5, 6, 10, 23, 24]. Momentum and Nesterov's Accelerated Gradient (NAG) accumulate a descent direction across iterations to alleviate zig-zagging and accelerate convergence [17, 19] . There are also meta-learning methods that allow networks to be trained jointly with their learning algorithm. Meta-methods may intelligently adjust hyperparameters like the learning rate, or learn the entire update term perhaps as a function of the batched gradient [1, 4]. Each of these techniques complement gradient descent to improve network learning and can be used in conjunction with our methods.

Prediction-correction methods are traditionally used in numerical analysis to integrate ordinary differential equations [22]. Since their inception, predictor-corrector methods have been used in a variety of fields that require optimization like theoretical study of chemical reactions and time-varying convex optimization [9, 21]. Prediction-correction has been incorporated into neural network training in the past by coevolving a pair of neural networks, a prediction network and a correction network [25, 26].

Scieur et al. propose a related learning algorithm to the one presented in this paper called Regularized Nonlinear Acceleration (RNA) [20]. RNA computes estimates of the optimum from a nonlinear average of a history of iterations produced by an optimization method like gradient descent. Like in RNA, the prediction step in PCGD is based on a history of parameter values obtained with gradient descent. However, our predictions use parameter specific linear regression rather than a nonlinear average of complete historical iterations. Making parameter specific predictions with linear regression allows our method to update predictions incrementally, which removes the need to keep all historical iterations relevant to a particular prediction. RNA must store the entire iteration history relevant to a particular prediction, which makes this method unfeasible for training large neural networks.

3 Methodology

PCGD uses best fit predictions and stochastic gradient descent in tandem. When estimating the trend in the network parameters through training, we will use fit

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

functions for which the least squares problem has a closed form solution using the normal equations. One could use more complex fit functions, but we want to avoid needing an extra iterative process. Using only least squares problems with closed form solutions to make parameter predictions also saves memory because they can be solved incrementally, avoiding the need to store a long history of network snapshots.

We will define the algorithm around the gradient descent iterations. We will make parameter predictions every p gradient descent iterations and collect snapshots of the network parameters every s^{th} gradient descent iteration where $p > s$ and $s|p$. Parameter predictions only consider the previous p/s network snapshots. Since $p > s$, only a sparse history of snapshots are considered. We'll call p the prediction increment and s the snapshot increment. For the remainder of this paper, the variables p and s will retain this definition.

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

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

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

We solve for prediction fit function variables A_{t+1} incrementally so as to minimize the extra storage required to perform PCGD. Fit function variables are updated at snapshot intervals. Let $\Theta_{t+1}^{(i)}$ denote the shorter matrix containing only the first i rows of Θ_{t+1}. Similarly, $J^{(i)}$ is the shorter matrix containing only the first i rows of J. When c snapshots have been recorded, we solve $J^{(c)}A_{t+1} = \Theta_{t+1}^{(c)}$ for the fit function variable matrix A_{t+1}; with c snapshots $J^{(c)}A_{t+1} = \Theta_{t+1}^{(c)}$

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

is a determined system. After this initial solve, only A_{t+1} must still be stored, $\Theta_{t+1}^{(c)}$ is no longer needed. At snapshot intervals $c+1$ through p/s we update the fit function variable matrix using the incremental least squares algorithm found in [3]. That is, for $i \in [c+1, p/s]$, we update,

$$A_{t+1} \leftarrow A_{t+1} + \boldsymbol{y}_i \left(\left(\boldsymbol{\theta}_{t+1}^{(i)} \right)^\top - \boldsymbol{j}_i^\top A_{t+1} \right) \tag{2}$$

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

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

By using an incremental least squares approach and solving for parameter specific best fit functions, we are able to conserve memory during training; without this approach one would need to store np/s parameter history values. This makes PCGD a feasible technique for training large networks provided c is small. Given the same history, RNA would solve for p/s coefficients for p/s entire network snapshots to obtain a nonlinear average of the whole snapshots [20]. Hence, RNA would require storing all np/s parameter history values. However, for the memory conservation afforded by incrementally updating fix functions, one pays a little extra work. Rather than solving for A_{t+1} directly, one must perform $p/s - c + 1$ incremental updates to A_{t+1}.

It should be noted that this is a general adaptation to stochastic gradient descent that is not specific to neural networks. This method may also appropriate for other high dimensional optimization problems.

4 Relationship to Nesterov's Accelerated Gradient

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

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

where A_t minimizes the L_2-norms of the columns of $JA_t - \Theta_t$. Here, $J \in \mathbb{R}^{(p/s)\times 2}$ has $\begin{bmatrix} 1^{i-1} \ 2^{i-1} \ \cdots \ (p/s)^{i-1} \end{bmatrix}^\top$ for its i^{th} column vector, and $\Theta_t \in \mathbb{R}^{(p/s)\times n}$ has

$\boldsymbol{\theta}_{t-p+is}^{\top}$ for its i^{th} row vector. With $p = 2$ and $s = 1$, this begins to look quite a bit like NAG algorithm which makes the update,

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

for specifically chosen series $\{\gamma_t\}_{t=0}^{\infty}$. With $l_t = 2 - \gamma_{t-1}$ these methods are identical. For continuously differentiable, smooth, convex loss functions NAG can achieve a global convergence rate of $O(1/t^2)$ [17, 2]. A natural extension of NAG incorporates a history of three points such that the update is

$$\begin{aligned} \lambda_t =& \left(1 + \sqrt{1 + 4\lambda_{t-r}^2}\right) \Big/ 2 \\ \boldsymbol{z}_t =& \begin{cases} \frac{\lambda_{t-1}}{\lambda_t}\boldsymbol{\theta}_t + \frac{(\lambda_t - 1)}{\lambda_t}\boldsymbol{\theta}_{t-r+1} - \frac{(\lambda_{t-1}-1)}{\lambda_t}\boldsymbol{\theta}_{t-r} & \text{if } t > r \\ \boldsymbol{\theta}_t & \text{otherwise} \end{cases} \\ \boldsymbol{\theta}_{t+1} =& \boldsymbol{z}_t - \epsilon \nabla L(\boldsymbol{z}_t) \end{aligned} \tag{3}$$

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

Theorem 1. *Let L be a convex, continuously differentiable and β-smooth function that admits a minimizer $\boldsymbol{\theta}^* \in \mathbb{R}^n$. Given an arbitrary initialization $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, for $T > r$ and $\epsilon = 1/\beta$, update scheme (3) satisfies,*

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

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

Corollary 1. *Let L be strongly convex with parameter $m > 0$, continuously differentiable and β-smooth function that admits a minimizer $\boldsymbol{\theta}^* \in \mathbb{R}^n$. Given an arbitrary initialization $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, for $T > r$ and $\epsilon = 1/\beta$, update scheme (3) satisfies,*

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

The order of r in the denominator on each side of the above inequality is the same. Hence, for $m = \beta$, $\min_{t \in \{T-r, \cdots, T\}} \{L(\boldsymbol{\theta}_{t+1}) - L(\boldsymbol{\theta}^*)\}$ converges at the same rate as NAG. The proof of Theorem 1 and Corollary 1 can be found in Appendix A [16].

In this well-behaved, theoretical environment, updating based on a linear combination of older values maintains the convergence rate of NAG. However, update method (3) is not practical for deep learning because it requires $r\times$ the memory to save a history of network parameter values. Instead, making parameter predictions every p^{th} iteration, as in update method (1), makes the additional memory requirement significantly more practical. In the setting of neural

network parameters, update method (1) has the capacity to outperform NAG. Considering an evenly distributed history of values extending further in the past allows one to de-noise trends. By incorporating a longer history, method (1) can afford to make predictions further into the future while minimizing additional memory requirements.

In comparison to NAG, employing update scheme (1) requires more memory for the fit function variables A_t, but performs less work as snapshot increment s and prediction increment p increase since fit function updates and parameter predictions happen less often. One must strike a balance though: for large p and large p/s one should be able to predict network parameters with more confidence provided the chosen fit function is well suited for the trend, but large p will exhibit delayed performance. Method (1) introduces a number of new hyperparameters that can be tuned for a particular task.

5 Experimental Results

The goal of our approach is to decrease the number of training epochs needed for a network to reach a particular testing accuracy. To test this, we ran experiments on the SVHN [18], and CIFAR10 [11] datasets using Krizhevsky's cuda-convnet with 4 hidden layers [12]. This net does not produce state-of-art accuracies for these datasets, but rather highlights the improvement seen by PCGD when compared to SGD. We implement our work in Caffe, which provides this architecture in their CIFAR10 "quick" example. We trained using batch size 100. Unless otherwise specified, hyperparameters and initialization distributions provided by Caffe's "quick" architecture are left unchanged. All experiments are run on the Bridges' NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center. Training is done with batched gradient descent using the cross-entropy loss function on the softmax of the output layer.

In this paper we will only use linear fit functions to make parameter predictions. That is, the fit function class is $f(\boldsymbol{a}, x) = a_1 + a_2 x$ and the number of fit function variables to solve for for each network parameter is $c = 2$. In this case, a network with n parameters requires storing an additional $2n$ values. If m is the maximum number of iterations we will train, p is the prediction increment and s is the snapshot increment, define $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, t) = b_1 + b_2 \, (t/p)^d$ where and $\boldsymbol{b}$ is chosen such that $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, 0) = p/s + u_1$ and $g_{(d,\boldsymbol{u})}(\boldsymbol{b}, m) = p/s + u_2$ for some $u_1, u_2 \in [0, 2p/s]$, $u_1 < u_2$. We chose our prediction length such that $l_t = g_{(d,\boldsymbol{u})}(\boldsymbol{b}, t)$. This means that at iteration p, PCGD tries to predict what the network weights will be at iteration $p + su_1$ and sets the weights to those predicted values. Similarly, at iteration m, PCGD *would* try to predict what the network weights would be at iteration $m + su_2$, but we do not make the last, or last few, predictions because immediately after predicting there is often a slight drop in accuracy that needs to be corrected by some gradient descent steps. This slight drop after predicting could be minimized by less aggressive predictions or better fit function choices, but we chose to simply leave out the last few predictions. It is a good idea to have u_1 small because parameter trends can alter and we do not want to be over-influenced by start-up trends.[1]

We will compare PCGD with NAG and SGD. We also consider a hybrid method combining NAG and PCGD, abbreviated as NAG-PCGD. To combine the two methods we nest NAG updates inside PCGD updates; the update scheme for NAG-PCGD is written out explicitly in Appendix B [16]. When training with PCGD and NAG-PCGD, we use prediction increment $p = 150$, snapshot increment $s = 15$ for all of our experiments. When plotting accuracy results, we will plot the maximum testing accuracy seen so far by that training iteration against iterations. While training, testing accuracy is usually noisy, which can obscure differences in performance when comparing different methods. Plotting the maximum testing accuracy seen so far displays these differences more clearly. There was no noticeable difference in the amount of noise seen in the testing accuracy for the various methods in our experiments.

5.1 SVHN

We experimented on the SVHN dataset with Krizhevsky's cuda-convnet [12]. The base learning rate was 0.001 and dropped by a factor of 10 after 4,000 iterations. Testing took place every 50 training iterations. When training with PCGD and NAG-PCGD, we use prediction length $l_t = g_{(6,[5,10])}(\boldsymbol{b}, t)$.

Figure 1 (Left) plots the maximum accuracy seen so far against iterations using standard SGD, NAG, PCGD and NAG-PCGD. Figure 1 (Right) plots the slopes of the curves in Figure 1 (Left) versus iteration. We show the iterations of steepest accuracy increase to highlight the difference in convergence rates of the various methods. NAG and NAG-PCGD initially increase at nearly the same rate which is $\approx 4\times$ faster than PCGD and SGD. Around iteration 450 PCGD leaves behind SGD, begins to catch up to NAG and eventually supersedes it. NAG-PCGD tends to hug the top of all the other curves exhibiting the benefits of both sub-methods. Confined to 2000 iterations, NAG-PCGD gives the best results. At iterations 4000 when the learning rate decreases by a factor of 10, there is another jump in accuracy where we can see the difference in convergence rates again on a smaller scale.

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

5.2 CIFAR10

We also trained Krizhevsky's cuda-convnet on the CIFAR10 for 195,000 iterations. The base learning rate was 0.001. We dropped the learning rate by a factor of 10 after 60,000 iterations and again after 125,000 iterations. Testing took place every 250 training iterations. We used $l_t = g_{(4,[5,10])}(\boldsymbol{b}, t)$ for our prediction length at prediction intervals.

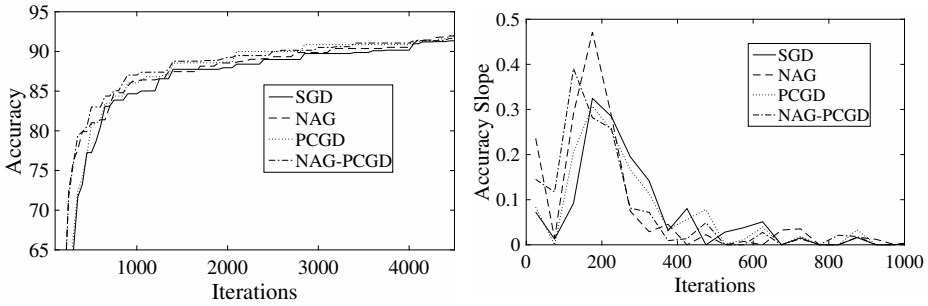

Fig. 1. (Left) Maximum accuracy results on the SVHN data set. Testing takes place every 50 training iterations (Right) Slope of Left Figure versus iterations.

Figure 2 (Left) shows maximum accuracy results through training using SGD, NAG, PCGD and NAG-PCGD. Again, we show only the iterations of steepest accuracy increase. Here, the testing increment is larger than our prediction increment which may hide any initial convergence advantage of NAG over PCGD. Given more time to excel, PCGD shows performance advantages over NAG; NAG does not even consistently outperform SGD per iteration. At any one time, NAG is at most 3.18% more accurate than SGD, PCGD is at most 3.91% more accurate than SGD, and NAG-PCGD is at most 6.49% more accurate than SGD.

Figure 2 (Right) shows, for a given accuracy, the percent of SGD iterations each method took to reach that accuracy. That is, if it took SGD x iterations to reach a particular accuracy for the first time, and PCGD took y iterations to reach that accuracy for the first time, then the value plotted for PCGD at that accuracy is $100 \times y/x$. This figure shows PCGD generally reaching particular accuracies before SGD and NAG-PCGD generally reaching accuracies before PCGD. SGD took 114,000 iterations to become 81.7% accurate. Training with NAG yielded 81.7% accuracy in 73% of the iterations required by SGD to reach this accuracy, training with PCGD yielded 81.7% accuracy in 56% of the iterations required by SGD and training with NAG-PCGD yielded 81.7% accuracy in 50% of the iterations required by SGD. That is, PCGD took only 77% of the iterations required by NAG to reach 81.7% accuracy.

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

6 Conclusion

We have developed a general adaptation to gradient descent and considered the impact in the case of training neural networks. Predictor-Corrector Gradient

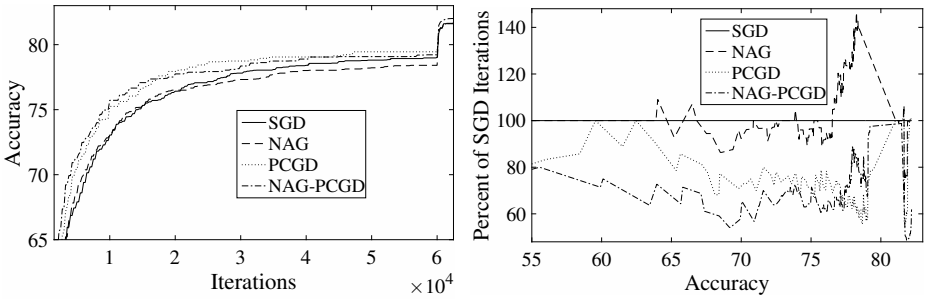

Fig. 2. Results on the CIFAR10 data set. (Left) Maximum Accuracy versus iterations. Testing takes place every 250 training iterations. (Right) Percent of SGD iterations each method took to reach a particular accuracy.

Descent reduces the number of iterations required to learn by incorporating traditional predictor-corrector inspired ideas into classic gradient descent.

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

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

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

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

References

1. Andrychowicz, M., et al.: Learning to learn by gradient descent by gradient descent. NIPS (2016)
2. Beck, A., et al.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. Siam Journal Imaging Sciences 2(1), 183–202 (2009)
3. Cassioli, A., et al.: An incremental least squares algorithm for large scale linear classification. European Journal of Operational Research 224(3), 560–565 (2013)
4. Daniel, C., et al.: Learning step size controllers for robust neural network training. AAAI (2016)
5. Dozat, T.: Incorporating Nesterov momentum into Adam. ICLR Workshop (2016)
6. Duchi, J., et al.: Adaptive subgradient methods for online learning and stochastic optimization. JMLR (2011)
7. Heeger, D.J.: Theory of cortical function. Proceedings of the National Academy of Sciences of the United States of America 114(8), 1773–1782 (2016)
8. Ho, Q., et al.: More effective distributed ml via a stale synchronous parallel parameter server. NIPS pp. 1223–1231 (2013)
9. Hratchian, H., et al.: Steepest descent reaction path integration using a first-order predictor-corrector method. The Journal of Chemical Physics 133(22) (2010)
10. Kingma, D., et al.: Adam: A method for stochastic optimization. ICLR (2015)
11. Krizhevsky, A.: Learning multiple layers of features from tiny images. Tech. rep., Computer Science, University of Toronto (2009)
12. Krizhevsky, A.: cuda-convnet. Tech. rep., Computer Science, University of Toronto (2012)
13. Krizhevsky, A., et al.: Imagenet classification with deep convolutional neural networks. NIPS pp. 1106–1114 (2012)
14. Luca, M.D., et al.: Optimal perceived timing: Integrating sensory information with dynamically updated expectations. Scientific Reports 6(28563) (2016)
15. Neelakantan, A., et al.: Adding gradient noise improves learning for very deep networks. arXiv:1511.06807 (2015)
16. Nesky, A., et al.: Training neural networks using predictor-corrector gradient descent: Appendix (2018), `http://www-personal.umich.edu/~anesky/PCGD_appendix.pdf`
17. Nesterov, Y.: A method of solving a convex programming problem with convergence rate o(1/sqr(k)). Soviet Mathematics Doklady 27, 372–376 (1983)
18. Netzer, Y., et al.: Reading digits in natural images with unsupervised feature learning. NIPS Workshop on Deep Learning and Unsupervised Feature Learning (2011)
19. Polyak, B.: Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics 4(5), 1–17 (1964)
20. Scieur, D., et al.: Regularized nonlinear acceleration. NIPS (2016)
21. Simonetto, A., et al.: Prediction-correction methods for time-varying convex optimization. IEEE Asilomar Conference on Signals, Systems and Computers (2015)
22. Süli, E., et al.: An Introduction to Numerical Analysis pp. 325–329 (2003)
23. Tieleman, T., et al.: Lecture 6a - rmsprop. COURSERA: Neural Networks for Machine Learning (2012)
24. Zeiler, M.D.: Adadelta: An adaptive learning rate method. arXiv:1212.5701 (2012)
25. Zhang, Y., et al.: Prediction-adaptation-correction recurrent neural networks for low-resource language speech recognition. arXiv:1510.08985 (2015)
26. Zhang, Y., et al.: Speech recognition with prediction-adaptation-correction recurrent neural networks. IEEE ICASSP (2015)