Diagonal Rescaling For Neural Networks

Jean Lafond, Nicolas Vasilache, Léon Bottou
Facebook AI Research, New York
lafond.jean@gmail.com, ntv@fb.com, leon@bottou.org

Abstract

We define a second-order neural network stochastic gradient training algorithm whose block-diagonal structure effectively amounts to normalizing the unit activations. Investigating why this algorithm lacks in robustness then reveals two interesting insights. The first insight suggests a new way to scale the stepsizes, clarifying popular algorithms such as RMSProp as well as old neural network tricks such as fanin stepsize scaling. The second insight stresses the practical importance of dealing with fast changes of the curvature of the cost.

1 Introduction

Although training deep neural networks is crucial for their performance, essential questions remain unanswered. Almost everyone nowadays trains convolutional neural networks (CNNs) using a canonical bag of tricks such as dropouts, rectified linear units (ReLUs), and batch normalization [Dahl et al., 2013, Ioffe and Szegedy, 2015]. Accumulated empirical evidence unambiguously shows that removing one of these tricks leads to less effective training.

Countless papers propose new additions to the canon. Following the intellectual framework set by more established papers, the proposed algorithmic improvements are supported by intuitive arguments and comparative training experiments on known tasks. This approach is problematic for two reasons. First, the predictive value of intuitive theories is hard to assess when they share so little with each other. Second, the experimental evidence often conflates two important but distinct questions: which learning algorithm works best when optimally tuned, and which one is easier to tune.

We initially hoped to help the experimental aspects by offering a solid baseline in the form of an efficient and well understood way to tune a simple stochastic gradient (SG) algorithm, hopefully with a performance that matches the canonical bag of tricks. To that effect, we consider reparametrizations of feedforward neural networks that are closely connected to the normalization of neural network activations [Schraudolph, 2012, Ioffe and Szegedy, 2015] and are amenable to zero overhead stochastic gradient implementations. Invoking the usual second order optimization arguments [Becker and LeCun, 1989, Ollivier, 2013, Desjardins et al., 2015, Marceau-Caron and Ollivier, 2016] leads to tuning the reparametrization with a simple diagonal or block-diagonal approximation of the inverse curvature matrix. The resulting algorithm performs well enough to produce appealing training curves and compete favorably with the best known methods. However this algorithm lacks robustness and occasionally diverges with little warning. The only way to achieve robust convergence seems to reduce the global learning rate to a point that negates its speed benefits.

Our critical investigation led to the two insights that constitute the main contributions of this paper. The first insight provides an elegant explanation for popular algorithms such as RMSProp [Tieleman and Hinton, 2012] and also clarifies well-known stepsize adjustments that were popular for the neural networks of the 1990s. The second insight explains some surprising aspects of batch normalization Ioffe and Szegedy [2015]. These two insights provide a unified perspective in which we can better understand and compare how popular deep learning optimization techniques achieve efficiency gains.

This document is organized as follows. Section 2 describes our reparametrization scheme for feedforward neural networks and discusses the efficient implementation of a SG algorithm. Section 3 revisits the notion of stepsizes when one approximates the curvature by a diagonal or block-diagonal matrix. Section 4.4 shows how fast curvature changes can derail many second order optimization methods and justify why it is attractive to evaluate curvature on the current minibatch as in batch-normalization.

2 Zero overhead reparametrization

This section presents our reparametrization setup for the trainable layers of a multilayer neural network. Consider a linear layer[1] with n inputs x_i and m outputs y_j

$$\forall j \in \{1 \dots m\} \qquad y_j = w_{0j} + \sum_{i=1}^{n} x_i w_{ij} \ . \tag{1}$$

Let E represent the value of the loss function for the current example. Using the notation $g_j = \frac{\partial E}{\partial y_j}$ and the convention $x_0 = 1$, we can write

$$\forall (i,j) \in \{0 \dots n\} \times \{1 \dots m\} \qquad \frac{\partial E}{\partial w_{ij}} = x_i g_j \ .$$

2.1 Reparametrization

We consider reparametrizations of (1) of the form

$$y_j = \beta_j \big(v_{0j} + \sum_{i=1}^{n} \alpha_i (x_i - \mu_i) v_{ij} \big) \ , \tag{2}$$

where v_{0j} and v_{ij} are the new parameters and μ_i, $\alpha_i > 0$, and $\beta_j > 0$ are constants that specify the exact reparametrization. The old parameters can then be derived from the new parameters with the relations

$$\begin{aligned} w_{ij} &= \alpha_i \beta_j v_{ij} \qquad (\text{ for } i = 1 \dots n. \text{ }) \\ w_{0j} &= \beta_j v_{0j} - \sum_{i=1}^{n} \mu_i w_{ij} = \beta_j v_{0j} - \sum_{i=1}^{n} \mu_i \alpha_i \beta_j v_{ij} \ . \end{aligned}$$

Using the convention $z_0 = 1$ and $z_i = \alpha_i (x_i - \mu_i)$, we can compactly write

$$y_j = \beta_j \sum_{i=0}^{n} v_{ij} z_i \ .$$

Running the SG algorithm on the new parameters amounts to updating these parameters by adding a quantity proportional to

$$\delta v_{ij} = \left\langle \frac{\partial E}{\partial v_{ij}} \right\rangle = \langle \beta_j g_j z_i \rangle \ ,$$

where the notation $\langle \dots \rangle$ is used to represent an averaging operation over a batch of examples. The corresponding modification of the old parameters is then proportional to

$$\begin{aligned} \delta w_{ij} = \alpha_i \beta_j \delta v_{ij} \ &= \ \big\langle \beta_j^2 g_j \ \alpha_i^2 (x_i - \mu_i) \big\rangle \\ \delta w_{0j} = \beta_j \delta v_{0j} - \sum_{i=0}^{n} \mu_i \delta w_{ij} \ &= \ \big\langle \beta_j^2 g_j \big\rangle - \sum_{i=1}^{n} \mu_i \delta w_{ij} \ . \end{aligned} \tag{3}$$

This means that we do not need to store the new parameters. We can perform both the forward and backward computations using the usual w_{ij} parameters, and use the above equations during the weight update. This approach ensures that we can easily change the constant α_i, μ_i, and β_j at any time without changing the function computed by the network.

Updating the weights using (3) is very cheap because we can precompute $\beta_j^2 g_j$ and $\alpha_i^2 (x_i - \mu_i)$ in time proportional to $n + m$. This overhead is negligible in comparison to the remaining computation which is proportional to nm.

[1] Appendix B discusses the case of convolutional layers.

2.2 Block-diagonal representation

The weight updates δw_{ij} described by equation (3) can also be obtained by pre-multiplying the averaged gradient vector $\langle \partial E/\partial w_{ij} \rangle = \langle g_j x_i \rangle$ by a specific block diagonal positive symmetric matrix. Each block of this pre-multiplication reads as

$$\begin{bmatrix} \delta w_{0j} \\ \delta w_{1j} \\ \vdots \\ \delta w_{nj} \end{bmatrix} = \beta_j^2 \begin{bmatrix} 1+\sum \alpha_i^2 \mu_i & -\alpha_1^2\mu_1 & \dots & -\alpha_n^2\mu_n \\ -\alpha_1^2\mu_1 & \alpha_1^2 & & 0 \\ \vdots & & \ddots & \\ -\alpha_n^2\mu_n & 0 & & \alpha_n^2 \end{bmatrix} \times \begin{bmatrix} \langle \partial E/\partial w_{0j} \rangle \\ \langle \partial E/\partial w_{1j} \rangle \\ \vdots \\ \langle \partial E/\partial w_{nj} \rangle \end{bmatrix}.$$

This rewrite makes clear that the reparametrization (2) is an instance of *quasi-diagonal* rescaling [Ollivier, 2013], with the additional constraint that, up to a scalar coefficient β_j^2, all the blocks of the rescaling matrix are identical within a same layer.

2.3 Choosing and adapting the reparametrization constants

Many authors have proposed *second order* stochastic gradient algorithms for neural networks [Becker and LeCun, 1989, Park et al., 2000, Ollivier, 2013, Martens and Grosse, 2015, Marceau-Caron and Ollivier, 2016]. Such algorithms rescale the stochastic gradients using a suitably constrained positive symmetric matrix. In all of these works, the key step consists in defining an approximation G of the curvature of the cost function, such as the Hessian matrix or the Fisher Information matrix, using ad-hoc assumptions that ensure that its inverse G^{-1} is easy to compute and satisfies the desired constraints on the rescaling matrix.

We can use the same strategy to derive sensible values for our reparametrization constants. Appendix A derives a block-diagonal approximation G of the curvature of the cost function with respect to the parameters v_{ij}. Each diagonal block G_j of this matrix has coefficients

$$[G_j]_{ii'} = \beta_j^2 \mathbb{E}\big[g_j^2\big] \times \begin{cases} \mathbb{E}\big[z_i^2\big] & \text{if } i = i', \\ \mathbb{E}[z_i]\,\mathbb{E}[z_{i'}] & \text{if } i \neq i', \end{cases} \tag{4}$$

where the expectation $\mathbb{E}[\cdot]$ is meant with respect to the distribution of the training examples. Choosing reparametrization constants μ_i, α_i, and β_j that make this surrogate matrix equal to the identity amounts to ensuring that a simple gradient step in the new parameters v_{ij} is equivalent to a second order step in the original parameters w_{ij}. This is achieved by choosing

$$\mu_i = \mathbb{E}[x_i] \qquad \alpha_i^2 = \frac{1}{\mathrm{var}[x_i]} \qquad \beta_j^2 = \frac{1}{\mathbb{E}\big[g_j^2\big]}\ . \tag{5}$$

It is not a priori obvious that we can continuously adapt the reparametrization constants on the basis of the observed statistics without creating potentially nefarious feedback loops in the optimization dynamics. On the positive side, it is well-known that pre-multiplying the stochastic gradients by a rescaling matrix provides the usual convergence guarantees if the eigenvalues of the rescaling matrix are upper and lower bounded by positive values [Bottou et al., 2016, §4.1], something easily achieved by adequately restricting the range of values taken by the reparametrization constants α_i^2, μ_i, and β_j^2. On the negative side, since the purpose of this adaptation is to make sure the rescaling matrix improves the convergence speed, we certainly do not want to see reparametrization constants hit their bounds, or, worse, bounce between their upper and lower bounds.

The usual workaround consists in ensuring that the rescaling matrix changes very slowly. In the case of our reparametrization scheme, after processing each batch of examples, we simply update online estimates of the moments

$$\begin{aligned} \mathtt{mx[i]} &\leftarrow \lambda\,\mathtt{mx[i]} + (1-\lambda)\,\langle x_i \rangle \\ \mathtt{mx2[i]} &\leftarrow \lambda\,\mathtt{mx2[i]} + (1-\lambda)\,\big\langle x_i^2 \big\rangle \\ \mathtt{mg2[j]} &\leftarrow \lambda\,\mathtt{mg2[j]} + (1-\lambda)\,\big\langle g_j^2 \big\rangle\,, \end{aligned}$$

with $\lambda \approx 0.95$, and we recompute the reparametrization constants (5). We additionally make sure that their values remain in a suitable range. This procedure is justified if we believe that the essential statistics of the x_i and g_j variables change sufficiently slowly during the optimization.

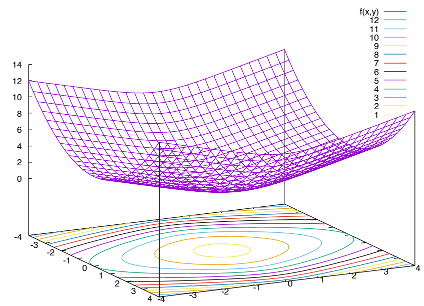

Figure 1: The function $F(w_1, w_2) = \frac{1}{2}w_1^2 + \log(e^{w_2} + e^{-w_2})$.

2.4 Informal comment about the algorithm performance

This algorithm performs well enough to produce appealing training curves and compete favorably with the best known methods (*at least for the duration of a technical paper*). The day-to-day practice suggests a different story which is both more important and difficult to summarize with experimental results. Finding a proper stepsize with plain SG is relatively easy because excessive stepsizes immediately cause a catastophic divergence. This is no longer the case with this proposed algorithm: many stepsizes appear to work efficiently, but occasionally cause divergence with little warning. The only way to achieve robust convergence seems to be to reduce the stepsize to a point that essentially negates the initial speed gain. This observation does not seem to be specific to our particular algorithm. For instance, Le Cun et al. [1998, §9.1] mention that, in practice, their diagonal rescaling method reduces the number of iterations by no more than a factor of three relative to plain SG, barely justifying the overhead.

3 Stepsizes and diagonal rescaling

The difficulty of finding good global stepsizes with second order optimization methods is in fact a well-known issue in optimization, only made worse by the stochastic nature of the algorithms we consider. After presenting a motivating example, we return to the definition of the stepsizes and develop an alternative formulation suitable for diagonal and block-diagonal rescaling approaches.

3.1 Motivating example

Figure 1 represents the apparently benign convex function

$$F(w_1, w_2) = \frac{1}{2}w_1^2 + \log(e^{w_2} + e^{-w_2}) \tag{6}$$

whose gradients and Hessian matrix respectively are

$$\nabla F(w_1, w_2) = \begin{bmatrix} w_1 \\ \tanh(w_2) \end{bmatrix} \qquad \nabla^2 F(w_1, w_2) = \begin{bmatrix} 1 & 0 \\ 0 & \cosh(w_2)^{-2} \end{bmatrix} .$$

Following Bottou et al. [2016, §6.5], assume we are optimizing this function with starting point $(3, 3)$. The first update moves the current point along direction $-\nabla F \approx [-3, -1]$ which unfortunately points slightly away from the optimum $(0, 0)$. Rescaling with the inverse Hessian yields a substantially worse direction $-(\nabla^2 F)^{-1}\nabla F \approx [-3, -101]$. The large second coefficient calls for a small stepsize. Using stepsize $\gamma \approx 0.03$ moves the current point to $(2.9, 0)$. Although the new gradient $\nabla F \approx [-2.9, 0]$ points directlty towards the optimum, the small stepsize that was necessary for the previous update is now ten times too small to effectively leverage this good situation.

We can draw two distinct lessons from this example:

a) A global stepsize must remain small enough to accomodate the most ill-conditioned curvature matrix met by the algorithm iterates. This is precisely why most batch second-order optimization techniques rely on line search techniques instead of fixing a single global stepsize [Nocedal and Wright, 2006], something not easily done in the case of a stochastic algorithm. Therefore it is desirable to automatically adjust the stepsize to account for the conditioning of the curvature matrix.

b) The objective function (6) is a sum of terms operating on separate subsets of the variables. Absent additional information relating these terms to each other, we can leverage this structural information by optimizing each term separately. Otherwise, as illustrated by our example, the optimization of one term can hamper the optimization of the other terms. Such functions have a block diagonal Hessian. Conversely, all functions whose Hessian is everywhere block diagonal can be written as such separated sums (Appendix C). Therefore, using a block-diagonal approximation of a curvature matrix is very similar to separately optimizing each block of variables.

3.2 Stepsizes for natural gradient

The classic derivation of the natural gradient algorithm provides a useful insight on the meaning of the stepsizes in gradient learning techniques [Amari and Nagaoka, 2000, Ollivier, 2013]. Consider the objective function $C(\mathbf{w}) = \mathbb{E}[E_\xi(\mathbf{w})]$, where the expectation is taken over the distribution of the examples ξ, and assume that the parameter space is equipped with a (Riemannian) metric in which the squared distance between two neighboring points $\mathbf{w}$ and $\mathbf{w} + \delta\mathbf{w}$ can be written as

$$D(\mathbf{w}, \mathbf{w} + \delta\mathbf{w})^2 = \delta\mathbf{w}^\top G(\mathbf{w})\, \delta\mathbf{w} + o(\|\delta\mathbf{w}\|^2)\ .$$

We assume that the positive symmetric matrix $G(\mathbf{w})$ carries useful information about the curvature of our objective function,[2] essentially by telling us how far we can trust the gradient of the objective function. This leads to iterations of the form

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \operatorname*{argmin}_{\delta\mathbf{w}} \left\{ \delta\mathbf{w}^\top \left\langle \nabla E(\mathbf{w}^t) \right\rangle \ \text{ subject to }\ \delta\mathbf{w}^\top G(\mathbf{w}^t)\, \delta\mathbf{w} \leq \eta^2 \right\}\ , \tag{7}$$

where the angle brackets denote an average over a batch of examples and where η represents how far we trust the gradient in the Riemannian metric. The classic derivation of the natural gradient reformulates this problem using by introducing a Lagrange coefficient $1/2\gamma > 0$,

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \operatorname*{argmin}_{\delta\mathbf{w}} \left\{ \delta\mathbf{w}^\top \left\langle \nabla E(\mathbf{w}^t) \right\rangle + \frac{1}{2\gamma} \delta\mathbf{w}^\top G(\mathbf{w}^t)\, \delta\mathbf{w} \right\}\ .$$

Solving for $\delta\mathbf{w}$ then yields the natural gradient algorithm

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \gamma\, G^{-1}(\mathbf{w}^t) \left\langle \nabla E(\mathbf{w}^t) \right\rangle\ . \tag{8}$$

It is often argued that choosing a stepsize γ is as good as choosing a trust region size η because every value of η can be recovered using a suitable γ. However the exact relation between γ and η depends on the cost function in nontrivial ways. The exact relation, recovered by solving $\delta\mathbf{w}^\top G(\mathbf{w}^t)^{-1}\delta\mathbf{w} = \eta^2$, leads to an expression of the natural gradient algorithm that depends on η instead of γ.

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \eta \frac{G^{-1}(\mathbf{w}^t) \left\langle \nabla E(\mathbf{w}^t) \right\rangle}{\sqrt{\left\langle \nabla E(\mathbf{w}^t) \right\rangle^\top G^{-1}(\mathbf{w}^t) \left\langle \nabla E(\mathbf{w}^t) \right\rangle}}\ . \tag{9}$$

Expression (9) updates the weights along the same direction as (8) but introduces an additional scalar coefficient that effectively modulates the stepsize in a manner consistent with Section 3.1.a. A similar approach was in advocated by Schulman et al. [2015] for the TRPO algorithm used in Reinforcement Learning. The next subsection shows how this approach changes when one considers a block-diagonal curvature matrix in a manner consistent with Section 3.1.b.

3.3 Stepsizes for block diagonal natural gradient

We now assume that $G(\mathbf{w})$ is block-diagonal. Let $\mathbf{w}_j$ represent the subset of weights associated with each diagonal block $G_{jj}(\mathbf{w})$. Following Section 3.1.b, we decouple the optimization of the variables associated with each block by replacing the natural gradient problem (7) by the separate problems

$$\forall j \qquad \mathbf{w}_j^{t+1} = \mathbf{w}_j^t + \operatorname*{argmin}_{\delta\mathbf{w}_j} \left\{ \delta\mathbf{w}_j^\top \left\langle \nabla_j E(\mathbf{w}_t) \right\rangle \ \text{ subject to }\ \delta\mathbf{w}_j^\top G_{jj}(\mathbf{w}^t)\, \delta\mathbf{w}_j \leq \eta^2 \right\}\ ,$$

where ∇_j represents the gradient with respect to w_j. Solving as above leads to

$$\forall j \qquad \mathbf{w}_j^{t+1} = \mathbf{w}_j^t + \eta \frac{G_{jj}^{-1}(\mathbf{w}^t) \left\langle \nabla_j E(\mathbf{w}^t) \right\rangle}{\sqrt{\left\langle \nabla_j E(\mathbf{w}^t) \right\rangle^\top G_{jj}^{-1}(\mathbf{w}^t) \left\langle \nabla_j E(\mathbf{w}^t) \right\rangle}}\ . \tag{10}$$

[2] This is why this document often refer to the Riemannian metric tensor $G(\mathbf{w})$ as the curvature matrix. This convenient terminology should not be confused with the notion of curvature of a Riemannian space.

This expression is in fact very similar to (9) except that the denominator is now computed separately within each block, changing both the length and the direction of the weight update.

It is desirable in practice to ensure that the denominator of expression (9) or (10) remains bounded away from zero. This is particularly a problem when this term is subject to statistical fluctuations induced by the choice of the batch of examples. This can be addressed using the relation

$$\begin{aligned} \big\langle \nabla_j E(\mathbf{w}^t) \big\rangle^\top G_{jj}^{-1}(\mathbf{w}^t) \left\langle \nabla_j E(\mathbf{w}^t) \right\rangle &\approx \mathbb{E}\big[\nabla_j E(\mathbf{w}^t)\big]^\top G_{jj}^{-1}(\mathbf{w}^t) \mathbb{E}\big[\nabla_j E(\mathbf{w}^t)\big] \\ &\leq \mathbb{E}\big[\nabla_j E(\mathbf{w}^t)^\top G_{jj}^{-1}(\mathbf{w}^t) \nabla_j E(\mathbf{w}^t)\big] \ . \end{aligned}$$

Further adding a small regularization parameter $\mu > 0$ leads to the alternative formulation

$$\forall j \qquad \mathbf{w}_j^{t+1} = \mathbf{w}_j^t + \eta \frac{G_{jj}^{-1}(\mathbf{w}^t) \left\langle \nabla_j E(\mathbf{w}^t) \right\rangle}{\sqrt{\mu + \mathbb{E}\big[\nabla_j E(\mathbf{w}^t)^\top G_{jj}^{-1}(\mathbf{w}^t) \nabla_j E(\mathbf{w}^t)\big]}} \ . \tag{11}$$

3.4 Recovering RMSprop

Let us first illustrate this idea by considering the Euclidian metric $G = I$. Evaluating the denominator of (11) separately for each weight and estimating the expectation $\mathbb{E}\big[(\nabla_j E)^2\big]$ with a running average

$$R_j^t = (1-\lambda) R_j^{t-1} + \lambda \left(\frac{\partial E}{\partial w_j} \right)^2 \ ,$$

yields the well-loved RMSProp weight update [Tieleman and Hinton, 2012]:

$$w_j^{t+1} = w_j^t - \frac{\eta}{\sqrt{\mu + R_j^t}} \left\langle \frac{\partial E}{\partial w_j} \right\rangle \ .$$

3.5 Recovering a well-known neural network trick

We now consider a neural network using the hyperbolic tangent activation functions as was fashionable in the 1990s [Le Cun et al., 1998]. Using the notations of Section 2, we consider block-diagonal curvature matrices whose blocks G_{jj} are associated to the weights $\mathbf{w}_j = (w_{0j} \dots w_{nj})$ of each unit j. Because this activation function is centered and bounded, it is almost reasonable to assume that the x_i have zero mean and unit variance. Proceeding with the approximations discussed in Appendix A, and further assuming the x_i are uncorrelated,

$$\big[G_{jj}\big]_{ii'} \ \approx\ \mathbb{E}\big[g_j^2\big]\, \mathbb{E}\Big[x_i x_{i'}\Big] \ \approx\ \begin{cases} \mathbb{E}\big[g_j^2\big] & \text{if } i = i' \\ 0 & \text{otherwise.} \end{cases}$$

We can then evaluate the denominator of (11), with $\mu = 0$, under the same approximations:

$$\sqrt{\frac{\mathbb{E}[\sum_{i=1}^n x_i g_j g_j x_i]}{\mathbb{E}\big[g_j^2\big]}} \ \approx\ \sqrt{\frac{\sum_{i=1}^n \mathbb{E}[x_i^2]\, \mathbb{E}\big[g_j^2\big]}{\mathbb{E}\big[g_j^2\big]}} \ =\ \sqrt{n} \ .$$

Although dividing the learning rate by the inverse square root of the number n of incoming connections (the fanin) is a well known trick for such networks [Le Cun et al., 1998, §4.7], no previous explanation had linked it to curvature issues.

Figure 2 (left) illustrates the effectiveness of this trick when training a typical convolutional network[3] on the CIFAR10 dataset. Although our network uses ReLU instead of hyperbolic tangent activations, the experiment shows the value of dividing the learning rates by $\sqrt{n \times s}$, where n represents the fanin and where the weight sharing count s is always 1 for a linear layer and can be larger for a convolutional layer (see Appendix B). In both cases we use mini-batches of 64 examples and select the global constant stepsize that yields the best training loss after 40 epochs.

[3] `https://github.com/soumith/cvpr2015/blob/master/Deep Learning with Torch.ipynb`

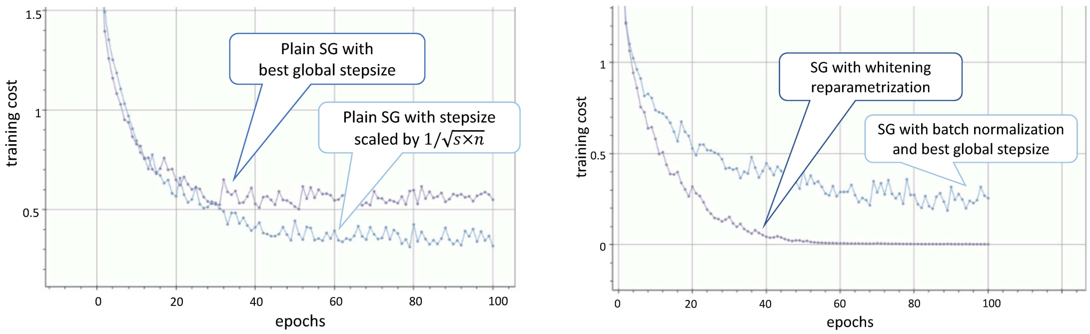

Figure 2: Training a typical convolutional network (C6(5x5)-P(2x2)-C16(5x5)-P(2x2)-F120-F84-F10) on the CIFAR10 dataset (60000 32×32 color images, 10 classes). Left: Stochastic gradient with global stepsize and with stepsize divided by $\sqrt{n \times s}$. Right: Stochastic gradient with batch normalization versus whitening reparametrization. Note that the vertical scales are different.

4 Whitening reparametrization

Since the zero-overhead reparametrization of Section 2 amounts to using a particular block-diagonal curvature matrix, we can apply the insight of the previous section and optimize the natural gradient problem within each block. Proceeding as in Section 3.5, we use the reparametrization constants

$$\mu_i = \mathbb{E}[x_i] \qquad \alpha_i^2 = \frac{1}{\mathrm{var}[x_i]} \qquad \beta_j^2 = \frac{1}{\sqrt{n \times s}}\,, \tag{12}$$

The only change relative to (5) consists in replacing the original $\beta_j^2 = 1/\mathbb{E}\big[g_j^2\big]$ by an expression that depends only on the geometry of the layer (the fanin n and the sharing count s). Meanwhile the constants α_i^2 and μ_i are recomputed after each minibatch on the basis of online estimates of the input moments as explained in Section 2.3.

An attentive reader may note that we should have multiplied (instead of replaced) the original β_j^2 by the scaling factor $1/\sqrt{n \times s}$. In practice, removing the $\mathbb{E}\big[g_j^2\big]$ term from the denominator makes the algorithm more robust, allowing us to use significantly larger global stepsizes without experiencing the occasionnal divergences that plagued our original algorithm (the cause of this behavior will become clearer in Section 4.4.)

4.1 Comparison with batch normalization

Batch normalization [Ioffe and Szegedy, 2015] is an obvious point of comparison for our reparametrization approach. Both methods attempt to normalize the distribution of certain intermediate results. However they do it in a substantially different way. The whitening reparametrization normalizes on the basis of statistics accumulated over time, whereas batch normalization uses instantaneous statistics observed on the current mini-batch. The whitening reparametrization does not change the forward computation. Under batch normalization, the output computed for any single example is affected by the other examples of the same mini-batch. Assuming that these examples are picked randomly, this amounts to adding a nontrivial noise to the computation, which can be both viewed as a nuisance and as a useful regularization technique.

4.2 Cifar10 experiments

Figure 2 (right plot) compares the evolution of the training loss of our CIFAR10 CNN using the whitening reparametrization or using batch normalization on all layers except the output layer. Whereas batch normalization shows a slight improvement over the unnormalized curves of the left plot, training with the whitening reparametrization quickly drives the training loss to zero.

From the optimization point of view, driving the training loss to zero is a success. From the machine learning point of view, this means that we overfit and must compensate by either adding explicit regularization or reducing the size of the network. As a sanity check, we have verified that we can recover the batch normalization testing error by adding L2 regularization to the network trained with the whitening reparametrization. The two algorithms then reduce the test error with similar rates.[4]

[4]Using smaller networks would of course yield better speedups. A better optimization algorithm can conceivably help reduce our reliance on vastly overparametrized neural networks [Zhang et al., 2017].

4.3 ImageNet experiments

In order to appreciate how the whitening reparametrization works at scale, we replicate the above comparison using the well known AlexNet convolutional network [Krizhevsky et al., 2012] trained on ImageNet (one million 224×224 training images, 1000 classes.)

The result is both disappointing and surprising. Training using only $100,000$ randomly selected examples in ImageNet reliably yields training curves similar to those reported in Figure 2 (right). However, when training on the full 1M examples, the whitening reparametrization approach performs very badly, not even reaching the best training loss achieved with plain stochastic gradient descent. The network appears to be stuck in a bad place.

4.4 Fast changing curvature

The ImageNet result reported above is surprising because the theoretical performance of stochastic gradient algorithm does not usually depend on the size of the pool of training examples. Therefore we spend a considerable time manually investigating this phenomenon.

The key insight was achieved by systematically comparing the actual statistics $\mathbb{E}[x_i]$ and $\mathrm{var}[x_i]$, estimated on a separate batch of examples, with those estimated with the slow running average method described in Section 2.3. Both estimation methods usually give very consistent results. However, in rare instance, they can be completely different. When this happens, the reparametrization constants α_i^2 and μ_i are off. This often leads to unreasonably large changes of the affected weights. When the bias of a particular unit becomes too negative, the ReLU activation function remains zero regardless of the input example, and no gradient signal can correct this in the future. In other words, these rare events progressively disable a significant fraction of the neural network units.

How can our slow estimation of the curvature be occasionally so wrong? The only possible explanation is that the curvature can occasionally change very quickly. How can the curvature change so quickly? With a homogenous activation function like the ReLU, one does not change the neural network output if we pick one unit, multiply its incoming weights by an arbitrary constant κ and divide its outgoing weights by the same constant. This means that the cost function in weight space is invariant along complex manifolds whose two-dimensional slices look like hyperbolas. Although the gradient of the objective function is theoretically orthogonal to these manifolds, a little bit of numerical noise is sufficient to cause a movement along the manifold when the stepsize is relatively large.[5] Changing the relative sizes of the incoming and outgoing weights of a particular unit can of course dramatically change the statistics of the unit activation.

This observation is important because most second-order optimization algorithms assume that the curvature changes slowly [Becker and LeCun, 1989, Martens and Grosse, 2015, Nocedal and Wright, 2006]. Batch normalization does not suffer from this problem because it relies on fresh mean and variance estimates computed on the current mini-batch. As mentioned in Section 2.3 and detailled in Appendix D, computing α_i and μ_i on the current minibatch creates a nefarious feedback loop in the training process. Appendix E describes an inelegant but effective way to mitigate this problem.

5 Conclusion

Investigating the robustness issues of a second-order block-diagonal neural network stochastic gradient training algorithm has revealed two interesting insights. The first insight reinterprets what is meant when one makes a block-diagonal approximation of the curvature matrix. This leads to a new way to scale the stepsizes and clarifies popular algorithms such as RMSProp as well as old neural network tricks such as fanin stepsize scaling. The second insight stresses the practical importance of dealing with fast changes of the curvature. This observation challenges the design of most second order optimization algorithms. Since much remains to be achieved to turn these insights into a solid theoretical framework, we believe useful to share both the path and the insights.

Acknowledgments Many thanks to Yann Dauphin, Yann Ollivier, Yuandong Tian, and Mark Tygert for their constructive comments.

[5] In fact such movements are amplified by second-order algorithms because the cost function has zero curvature in directions tangent to these manifolds. This is why we experienced so many problems with the $\beta_j^2 = 1/\mathbb{E}\big[g_j^2\big]$ scaling suggested by the naïve second-order viewpoint.

References

Sun-Ichi Amari and Hiroshi Nagaoka. *Methods of Information Geometry*. Oxford University Press, Oxford, 2000.

S. Becker and Y. LeCun. Improving the convergence of back-propagation learning with second-order methods. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proc. of the 1988 Connectionist Models Summer School*, pages 29–37, San Mateo, 1989. Morgan Kaufman.

L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *ArXiv e-prints*, June 2016.

George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013*, pages 8609–8613, 2013.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. In *Advances in Neural Information Processing Systems 28*, pages 2071–2079, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456, 2015.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, 2012.

Y. Le Cun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag, 1998.

Yann Le Cun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag, 1998.

Gaétan Marceau-Caron and Yann Ollivier. Practical Riemannian neural networks. *ArXiV CoRR*, abs/1602.08007, 2016. URL `http://arxiv.org/abs/1602.08007`.

James Martens. New insights and perspectives on the natural gradient method. *ArXiV CoRR*, abs/1412.1193, 2014. URL `http://arxiv.org/abs/1412.1193`.

James Martens and Roger B. Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, pages 2408–2417, 2015.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer New York, Second edition, 2006.

Yann Ollivier. Riemannian metrics for neural networks. *ArXiV CoRR*, abs/1303.0818, 2013. URL `http://arxiv.org/abs/1303.0818`.

Hyeyoung Park, Sun-ichi Amari, and Kenji Fukumizu. Adaptive natural gradient learning algorithms for various stochastic models. *Neural Networks*, 13(7):755–764, 2000.

Nicol N. Schraudolph. Centering neural network gradient factors. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2012.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*, pages 1889–1897, 2015.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5. RMSPROP: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Representation Learning (ICLR 2017)*, 2017. Also arXiV CoRR abs/1611.03530.

Appendices

A Derivation of the curvature matrix

For the sake of simplicity, we only take into account the parameters $\mathbf{v} = (\dots v_{ij} \dots)$ associated with a particular linear layer of the network (hence neglecting all cross-layer interactions). Each example is then represented by the layer inputs x_i and by an additional variable ξ that encode any relevant information not described by the x_i. For instance ξ could represent a class label. We then assume that the cost function associated with a single example has the form

$$E(\mathbf{v}; \xi, x_1 \dots x_n) = -\log\big(\varphi(\xi, y_1 \dots y_m)\big) = -\log\left(\varphi\left(\xi, \dots \beta_j \sum_{i=0}^{n} v_{ij} x_i \dots\right)\right),$$

where the function φ encapsulates all the layers following the layer of interest as well as the loss function. This kind of cost function is very common when the quantity φ can be interpreted as the probability of some event of interest.

The optimization objective $C(\mathbf{v})$ is then the expectation of E with respect to the variables ξ and x_i,

$$C(\mathbf{v}) = \mathbb{E}[E(\xi, x_1 \dots x_n)]\,.$$

Its derivatives are

$$\frac{\partial C}{\partial v_{ij}} \;=\; \mathbb{E}\left[-\frac{1}{\varphi}\frac{\partial \varphi}{\partial y_j}\beta_j z_i\right] \;=\; \mathbb{E}[g_j \beta_j z_i]$$

and the coefficients of its Hessian matrix are

$$\frac{\partial^2 C}{\partial v_{ij} \partial v_{i'j'}} \;=\; \mathbb{E}\left[\left(\frac{1}{\varphi^2}\frac{\partial \varphi}{\partial y_j}\frac{\partial \varphi}{\partial y_{j'}} - \frac{1}{\varphi}\frac{\partial^2 \varphi}{\partial y_j y_{j'}}\right)\beta_j \beta_{j'} z_i z_{i'}\right]\,.$$

Our first approximation consists in neglecting all the terms of the Hessian involving the second derivatives of φ, leading to the so-called *Generalized Gauss-Newton* matrix G [Bottou et al., 2016, §6.2] whose blocks $G_{jj'}$ have coefficients

$$\big[G_{jj'}\big]_{ii'} \;=\; \mathbb{E}\left[\frac{1}{\varphi^2}\frac{\partial \varphi}{\partial y_j}\frac{\partial \varphi}{\partial y_{j'}}\beta_j \beta_{j'} z_i z_{i'}\right] \;=\; \beta_j \beta_{j'} \mathbb{E}[g_j g_{j'} z_i z_{i'}]$$

Interestingly, this matrix is exactly equal to a well known approximation of the Fisher information matrix called the *Empirical Fisher* matrix [Park et al., 2000, Martens, 2014].

We then neglect the non-diagonal blocks and assume that the squared gradients g_j^2 are not correlated with either the layer inputs x_i or their cross products $x_i x_{i'}$. See [Desjardins et al., 2015] for a similar approximation. Recalling that $z_0 = 1$ is not correlated with anything by definition, this means that the g_j^2 is not correlated with $z_i z_{i'}$ either.

$$\big[G_{jj}\big]_{ii'} \;=\; \beta_j^2 \mathbb{E}\big[g_j^2\big]\, \mathbb{E}[z_i z_{i'}]\,.$$

Further assuming that the layer inputs x_i are also decorrelated leads to our final expression

$$\big[G_{jj}\big]_{ii'} \;=\; \beta_j^2 \mathbb{E}\big[g_j^2\big] \times \begin{cases} \mathbb{E}\big[z_i^2\big] & \text{if } i = i', \\ \mathbb{E}[z_i]\,\mathbb{E}[z_{i'}] & \text{otherwise.} \end{cases}$$

The validity of all these approximations is of course questionable. Their true purpose is simply to make sure that our approximate curvature matrix G can be made equal to the identity with a simple choice of the reparametrization constants, namely,

$$\mu_i = \mathbb{E}[x_i] \qquad \alpha_i^2 = \frac{1}{\mathrm{var}[x_i]} \qquad \beta_j^2 = \frac{1}{\mathbb{E}\big[g_j^2\big]}\,.$$

B Reparametrization of convolutional layers

Convolutional layers can be reparametrized in the same manner as linear layers (Section 2) by introducing additional indices u and v to represent the two dimensions of the image and kernel coordinates. Equations (1) and (2) then become

$$\begin{aligned} y_{ju_1u_2} &= w_{0j} + \sum_{i=1}^{n} \sum_{v_1v_2} x_{i(u_1+v_1)(u_2+v_2)}\, w_{ijv_1v_2} \\ &= \beta_j \left(v_{0j} + \sum_{i=1}^{n} \sum_{v_1v_2} \alpha_i \big(x_{i(u_1+v_1)(u_2+v_2)} - \mu_i \big)\, v_{ijv_1v_2} \right) , \end{aligned}$$

and the derivative of the loss E with respect to a particular weight involves a summation over all the terms involving that weight:

$$\frac{\partial E}{\partial v_{ijv_1v_2}} = \beta_j \sum_{u_1u_2} g_{ju_1u_2}\, z_{i(u_1+v_1)(u_2+v_2)} \, .$$

Following Appendix A, we write the blocks $G_{jj'}$ of the generalized Gauss Newton matrix G,

$$[G_{jj'}]_{iv_1v_2,i'v_1'v_2'} = \mathbb{E}\left[\frac{\partial E}{\partial v_{ijv_1v_2}} \frac{\partial E}{\partial v_{i'j'v_1'v_2'}} \right] .$$

Obtaining a convenient approximation of G demands questionable assumptions such as neglecting nearly all off-diagonal terms, and nearly all possible correlations involving the z and g variables. This leads to the following choices for the reparametrization constants, where the expectations and variances are also taken across the image dimension subscripts ("$\bullet$") and where the constant s counts the number of times each weight is shared, that is, the number of applications of the convolution kernel in the convolutional layer.

$$\mu_i = \mathbb{E}[x_{i\bullet\bullet}] \qquad \alpha_i^2 = \frac{1}{\mathrm{var}[x_{i\bullet\bullet}]} \qquad \beta_j^2 = \frac{1}{s\,\mathbb{E}[g_{j\bullet\bullet}^2]} \, .$$

C Coordinate separation

It is obvious that a twice differentiable function

$$f \;:\; (x_1 \dots x_k) \in \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \;\longmapsto f(x_1, \dots, x_k) \in \mathbb{R}$$

that can be written as a sum

$$f(x_1 \dots x_k) = f_1(x_1) + \cdots + f_k(x_k) \tag{13}$$

has a block diagonal Hessian everywhere, that is,

$$\forall (x_1 \dots x_k) \in \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \quad \forall i \neq j \quad \frac{\partial^2 f}{\partial x_i \partial x_j} = 0 \, . \tag{14}$$

Conversely, assume the twice differentiable function f satisfies (14), and write

$$\begin{aligned} f(x_1 \dots x_k) - f(0 \dots 0) &= \sum_{i=1}^{k} f(x_1 \dots x_i, 0 \dots 0) - f(x_1 \dots x_{i-1}, 0 \dots 0) \\ &= \sum_{i=1}^{k} \int_0^1 x_i^\top \frac{\partial f}{\partial x_i}(x_1 \dots x_{i-1}, tx_i, 0 \dots 0)\, dt \, . \end{aligned}$$

Then observe

$$\begin{aligned} &\frac{\partial f}{\partial x_i}(x_1 \dots x_{i-1}, r, 0, \dots 0) - \frac{\partial f}{\partial x_i}(0 \dots 0, r, 0 \dots 0) \\ &\quad = \int_0^1 \sum_{j=1}^{i-1} x_j^\top \frac{\partial^2 f}{\partial x_j \partial x_i}(tx_1 \dots tx_{i-1}, r, 0 \dots 0)\, dt \;=\; 0 \, . \end{aligned}$$

Therefore property (13) is true because

$$f(x_1 \dots x_k) = f(0 \dots 0) + \sum_{i=1}^{k} \int_0^1 x_i^\top \frac{\partial f}{\partial x_i}(0 \dots 0, tx_i, 0, \dots 0)\, dt \, .$$

D Coupling effects when adapting reparametrization constants

The reparametrization constants suggested by (5) and (12) are simple statistical measurements on the network variables. It is tempting use to directly compute estimates $\hat{\alpha}_i$, $\hat{\mu}_i$, and $\hat{\beta}_j$ on the current mini-batch in a manner similar to batch renormalization.

Unfortunately these estimates often combine in ways that create unwanted biases. Consider for instance the apparently benign case where we only need to compute an estimate $\hat{\mu}_i$ because an oracle reveals the exact values of α_i and β_j. Replacing μ_i by its estimate $\hat{\mu}_i$ in the update equations (3) gives the actual weight updates $\widehat{\delta w_{ij}}$ performed by the algorithm. Recalling that $\hat{\mu}_i$ is now a random variable whose expectation is μ_i, we can compare the expectation of the actual weight update $\mathbb{E}\left[\widehat{\delta w_{0j}}\right]$ with the ideal value $\mathbb{E}[\delta w_{0j}]$.

$$
\begin{aligned}
\mathbb{E}\left[\widehat{\delta w_{0j}}\right] &= \mathbb{E}\left[\beta_j^2 g_j \left(1 - \sum_i \alpha_i^2 \hat{\mu}_i (x_i - \hat{\mu}_i)\right)\right] \\
&= \beta_j^2 \left(1 - \sum_i \alpha_i^2 \left(\mathbb{E}[\hat{\mu}_i x_i g_j] - \mathbb{E}[\hat{\mu}_i^2 g_j]\right)\right) \\
&= \mathbb{E}[\delta w_{0j}] + \sum_i \beta_j^2 \alpha_i^2 \left(\operatorname{var}[\hat{\mu}_i]\, \mathbb{E}[g_j] + \operatorname{cov}[\hat{\mu}_i^2, g_j] - \operatorname{cov}[\hat{\mu}_i, x_i g_j]\right) \ .
\end{aligned}
$$

This derivation reveals a systematic bias that results from the nonzero variance of $\hat{\mu}_i$ and its potential correlation with other variables. In practice, this bias is more than sufficient to severely disrupt the convergence of the stochastic gradient algorithm.

E Mitigating fast curvature change events

Fast curvature changes mostly happens during the first phase of the training process and disappears when the training loss stabilizes. For ImageNet, we were able to mitigate the phenomenon by using batch-normalization during the first epoch then switching to the whitening reparametrization approach for the remaining epochs (Figure 3.)

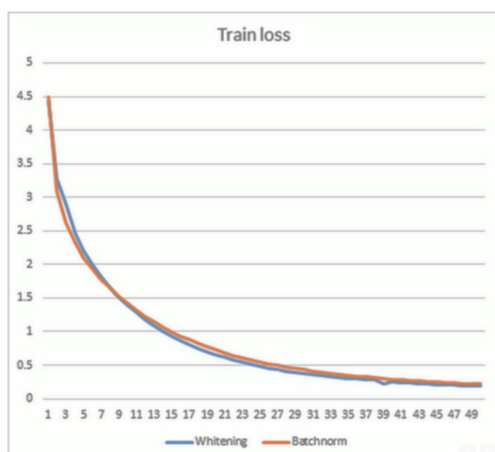

Figure 3: Mitigating fast curvature change events by using batch-normalization during the first epoch then either switching to the whitening reparametrization (blue curve) or keeping the batch normalization (blue curve). Although both methods appear similar in terms of number of epochs, the whitening reparametrization implementation is faster than the optimized batch normalization implementation. Note that the training loss in this curve was estimated after each epoch by performing a full sweep on the training data (unlike figure 2 which plots an estimate of the loss computed while training.)